NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

EXPLORATION AND VALIDATION OF THE SDHASH PARAMETER SPACE

by

Michael R. McCarrin

June 2013

Thesis Advisor:	Joel D. Young
Thesis Co-Advisor:	Simson L. Garfinkel

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–6–2013	Master's Thesis	2011-06-27—2013-6-21

4. TITLE AND SUBTITLE

EXPLORATION AND VALIDATION OF THE SDHASH PARAMETER SPACE

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Michael R. McCarrin

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Naval Postgraduate School
Monterey, CA 93943

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Department of the Navy

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

13. SUPPLEMENTARY NOTES

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.IRB Protocol Number: N/A

14. ABSTRACT

Cryptographic hashes are commonly used to aid in the examination of digital evidence by providing a method of rapidly identifying targeted content (e.g., incriminating materials) in large quantities of data. Because only exact matches can be detected, this method is easily defeated by even the smallest modification to the data. Approximate matching techniques maintain nearly the speed and space efficiency advantages of cryptographic hashes, while offering a more robust scheme for detecting similar objects. We seek to validate design choices in *sdhash*, the current state-of-the-art approximate matching algorithm, and suggest alternatives where appropriate. In addition, we clarify various nuances regarding the interpretation of its output so that it can be more effectively applied to forensic analysis. To this end, we provide a detailed analysis of *sdhash*'s behavior across a variety of relevant scenarios using the FRASH testing framework, and propose strategies for extracting more relevant and granular feedback.

15. SUBJECT TERMS

Digital Forensics, Digital Fingerprinting, Approximate Matching, Fuzzy Hashing, Similarity Digests, sdhash, FRASH

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER *(include area code)*
Unclassified	Unclassified	Unclassified	UU	125	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

EXPLORATION AND VALIDATION OF THE SDHASH PARAMETER SPACE

Michael R. McCarrin
Civilian, Department of the Navy
B.A., Brown University, 2002
M.F.A., San Francisco State University, 2010

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2013

Author: Michael R. McCarrin

Approved by: Joel D. Young
Thesis Advisor

Simson L. Garfinkel
Thesis Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Cryptographic hashes are commonly used to aid in the examination of digital evidence by providing a method of rapidly identifying targeted content (e.g., incriminating materials) in large quantities of data. Because only exact matches can be detected, this method is easily defeated by even the smallest modification to the data. Approximate matching techniques maintain nearly the speed and space efficiency advantages of cryptographic hashes, while offering a more robust scheme for detecting similar objects. We seek to validate design choices in *sdhash*, the current state-of-the-art approximate matching algorithm, and suggest alternatives where appropriate. In addition, we clarify various nuances regarding the interpretation of its output so that it can be more effectively applied to forensic analysis. To this end, we provide a detailed analysis of *sdhash*'s behavior across a variety of relevant scenarios using the FRASH testing framework, and propose strategies for extracting more relevant and granular feedback.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgements

This project could not have been completed without the help and support of many. I am extremely grateful for the guidance and dedication of my thesis advisors. I owe thanks to Dr. Simson Garfinkel for his cheerful criticism and advice, and for an untiring willingness to share the depth of his expertise, even on short notice. And without the persistent encouragement and determined open-source evangelism of Dr. Joel Young, my transition from the humanities to computer science would never have occurred. I am at a loss to express my gratitude for his unsettling attention to detail and constant insight, as well as his patient coaching and friendship.

I would like to thank my parents for always supporting me in everything, for helping whenever I asked, and for enduring my state of total distraction in the months leading up to the completion of this work.

Thank you, Thao, for keeping me company, keeping me sane, and listening to every detail.

Thanks also to Cynthia Irvine for her commitment to the Scholarship for Service program at NPS, without which I would not have met my many extraordinary classmates. Finally, thank you to my classmates and friends for hundreds of miscellaneous favors, for sharing in challenges and successes, and for solidarity through many sleepless nights in the windowless rooms of DEEP lab.

Partial support for this work was provided by the National Science Foundation's CyberCorps®: Scholarship for Service (SFS) program under Award No. 0912048. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Faced with limited time and overwhelming quantities of data, forensic examiners must be able to quickly and automatically correlate similar data objects without relying on a file system or program-specific APIs. Two among many possible use cases are as follows:

1. An examiner is targeting a specific, known data object or set of objects and wants to search a large amount of data to detect the presence of any similar objects. Applications include triage, locating incriminating material and detecting data exfiltration.
2. An examiner has a set of data objects and needs to determine which, if any, are similar to each other, and to prioritize the results by degree of similarity. Applications include characterizing newly acquired data with respect to current holdings and discovering correlations within large, diverse data sets.

Currently, many examiners leverage cryptographic hashes of files to aid in the accomplishment of these tasks. In the first case, cryptographic hashes can speed analysis of digital evidence both by filtering out data that is known to be irrelevant and identifying files that match content of interest (e.g., incriminating materials). In the second case, they provide an effective means of locating duplicate data. Because cryptographic hashes can only detect exact matches, a more flexible approach is needed.

Approximate matching algorithms attempt to provide such an approach by maintaining nearly the speed and space efficiency of cryptographic hash comparisons while offering a more robust scheme for correlating data objects that exhibit high-level (semantic) similarities. Roussev's *sdhash* utility [1] is emerging as a promising solution, combining advanced hashing techniques with an entropy-based feature-selection algorithm.

Roussev's algorithm contains a number of tunable parameters that could affect its performance and accuracy. No extensive study of the impact of these parameter choices is currently available. Determining optimal settings requires a consistent, representative testing framework, and clearly defined goals for the algorithm's behavior. Breitinger has made strides in this direction with the development of the FRASH utility for measuring the performance of approximate matching algorithms [2]. Several refinements to this framework are needed to allow for a meaningful comparison of different parameterizations of *sdhash*.

With respect to defining expected behavior, much work remains: no widespread agreement has been reached as to what the determination of "similarity" entails, and it is often unclear what the output of an approximate matching algorithm is claiming about the relationship between the objects it compares. This question must be answered before the consequences of parameter choices can be assessed, since any effort to interpret FRASH's evaluation of *sdhash* presupposes an understanding of the implications of the "similarity scores" that the algorithm produces.

For a given pair of files, the *sdhash* algorithm produces a score between 0 and 100, but this represents neither a percentage match nor a degree of confidence. They serve as an indicator of what proportion of material in the smaller of two compared objects can be matched to material in the larger. This formulation gives rise to nuanced behavior. For example:

- Because comparisons are always made small-to-large, comparing a file fragment to a large data object in which it is completely contained will yield an extremely high score, regardless of the larger object's size, since all material in the smaller object will be matched.
- By the same reasoning, reducing the size of the fragment will have almost no effect on the score as long as all parts of the remaining smaller object can still be found in the larger.
- Embedding the same fragment in two otherwise dissimilar objects and comparing them will give a score proportional to the fraction of matching material to total material in the smaller file. It follows that removing dissimilar material from the smaller object will increase the score, but increasing the amount of dissimilar material in the larger object will have no effect.

The complexity of this metric presents difficulties for forensic examiners attempting to draw conclusions about the objects being compared. Roussev addresses this problem by recommending a similarity score threshold of 21 [1], above which the compared object should be considered similar. This approach effectively reduces the tool's output to a binary classification. Though easier to understand, this method of interpreting the score introduces at least two problems.

First, the ideal value of this score threshold may depend on the particular data or task at hand. Roussev argues convincingly that over 99.9% of similar objects will be given a score of 22 or higher in a sample of mixed data types [1]. In contrast, his choice of 21 as a recommended value for eliminating false positives occurs as a compromise intended to produce the most consistent results across different file types and quantities of common material. Using a different sample set [3], he observes that a threshold as low as 5 can be used for HTML and text files with very little increase in the false positive rate, and by using this threshold he successfully identifies a

significant number of true positives that would be missed with a threshold of 21–22.

Second, restricting the interpretation of the score to a binary determination discards potentially useful information. No claims are made with regard to the degree of similarity between the two objects compared, despite the clear utility of such granular feedback.

An in-depth analysis of the implications of *sdhash*'s similarity scores has the potential to address these problems by making a direct interpretation of the scores more accessible, thereby taking fuller advantage of algorithm's properties, and allowing examiners to glean more information. Progress in this regard also has the benefit of providing clearer guidance regarding the choice of parameter values.

1.1 Research Objectives

This thesis seeks to validate design choices in *sdhash*, suggesting alternatives where appropriate, with the aim of enhancing the clarity and depth of information that can be obtained from its output. In addition, we investigate the extent to which the algorithm's similarity scores can be leveraged to describe the relationship between the objects compared. To accomplish these objectives, we first perform a static review of *sdhash*'s method of feature selection and similarity score calculation in order to identify several key parameters for testing. We then develop a customized version of *sdhash* that allows run-time modification of these parameters, and we test each variation using the FRASH testing framework, after making several alterations to said framework to permit it to handle a wider variety of algorithm behavior and to include enhanced false positive detection.

Once we have characterized the effects of varying each of our parameters, we endeavor to identify which sets of test results—and therefore which parameters settings—correspond to optimal behavior. This is not simply a matter of producing higher scores (which would be trivial); rather, the output must respond appropriately in accordance with what is being tested.

To our knowledge, no accepted standard of correctness currently exists against which *sdhash*'s behavior can be measured. For this reason we attempt to describe the expected output under the conditions generated by each test, and to use this as a criteria against which the effectiveness of the various parameter choices can be evaluated.

1.2 Significant Findings and Contributions

We propose disambiguating *sdhash*'s similarity scores by dividing the concept of "similarity" into two separate properties—"containment" and "commonality"—corresponding to the two use cases discussed. Containment describes the extent to which material in the smaller of two objects can be found in the larger. This represents *sdhash*'s current behavior, and the main contribution of the term is to make that behavior explicit. In contrast, commonality corresponds more closely with intuitive expectations of a similarity score. It measures the extent to which two objects are composed of common material—and should degrade gracefully when dissimilar material is added or similar material is reduced. We argue that *sdhash* could be easily modified to provide a score that meets these criteria, and we recommend modifying *sdhash* to allow an examiner to explicitly request that the output be a measurement of either commonality or containment, according to what is needed.

As the sizes of the compared objects approach each other, containment and commonality become indistinguishable. When this is the case, *sdhash*'s current scoring system behaves exactly like a commonality score, and is in fact more naturally interpreted as measuring commonality because the natural language definition of containment implies different-sized objects. This relationship allows us to evaluate parameter choices with respect to both properties using only *sdhash*'s current scoring system by examining the relative sizes of objects involved in each test and categorizing them in terms of their "intent" (i.e., whether they are best described as measuring commonality or containment). Having established these test categories, we recommend parameter choices that optimize the tool for each of these two suggested modes.

Pursuit of the above-mentioned goals lead to a number of secondary contributions. Chief among these was a careful review of the FRASH framework and numerous suggestions for enhancement, the majority of which were adopted into FRASH version 2.0 (though unfortunately due to time restrictions this new version could not be incorporated into our experiments). We also recommend a minor bug fix for *sdhash* which we believe will improve its speed on certain inputs.

1.3 Thesis Structure

The remainder of this thesis is organized as follows: Chapter 2 provides an overview of related work in approximate matching algorithms, as well as a description of pertinent concepts and a brief discussion of the FRASH testing framework. Chapter 3 enumerates the experiments

performed and gives a detailed description of the purpose and methodology of each. Chapter 4 documents and analyzes experimental results, and Chapter 5 presents conclusions and suggests opportunities for future research.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Related Work

As suggested by its name, *sdhash*'s approximate matching algorithm has its foundation in traditional hashing techniques. This chapter begins with a brief description of this foundation, then proceeds to outline the major innovations that have allowed approximate matching algorithms to advance beyond the limitations of basic hash comparison. Finally, we discuss the problem of measuring algorithm effectiveness and a recently proposed solution.

Broadly speaking, progress in the domain of approximate matching can be characterized as a series of increasing departures from strict matching criteria that find only identical objects, toward more nuanced comparisons that produce matches across a range of similar objects. More precisely, the algorithms relevant to this discussion rely on binary-level properties of data to sort objects into equivalence classes such that members of the same class exhibit common high-level characteristics; the general trajectory of progress among these algorithms can be summarized as a gradual refining of these equivalence classes. In the simplest case, e.g., a hashing scheme where only exact matches are identified, each object's equivalence class contains only itself. More sophisticated techniques endeavor to preserve the time and space advantages of traditional hashing while widening the associated equivalence classes (and keeping false positives to a minimum).

2.1 Terminology

Though implementation details vary, all matching algorithms discussed here share a common notion of both "feature" and "signature." The former term refers broadly to any attribute of a data object that might be selected as a basis of comparison (*sdhash*, for example, uses features consisting of 64-byte strings of data). The latter is a set of one or more features, typically compressed with a hash function, that identifies a data object. These concepts merit special attention because variations in their implementation determine many of the key differences between various matching algorithms. Moreover, *sdhash*'s signatures are represented by what Roussev refers to as a "similarity digest," a data structure composed of a series of concatenated Bloom filters; for this reason we begin with a brief description of Bloom filters and their basic properties.

2.2 Bloom Filters

First proposed by Burton Bloom [4], a Bloom filter is a compact data structure for representing elements in a set. Its salient properties include considerable space and time advantages gained in exchange for an arbitrarily small probability of encountering a false positive when the presence of an element is queried. False negatives are not possible.

A Bloom filter supports inserting elements and querying for them. Removing elements is not possible without auxiliary mechanisms. Before any elements are inserted, a Bloom filter is identical to an array of length m with all values set to zero. Elements are added by hashing them with k independent hash functions that map to indices within the bounds of the array. Each of these k indices is then set to 1. This is repeated for each element in the set.

The mechanism for querying elements is similar to adding them. The element is first hashed using the same k independent hash functions. The value at each of the resulting k indices is then tested to see whether it has been set to 1. A major advantage of Bloom filters is that this check can be performed in constant time. If one of the values is set to 0, we know with certainty that the element has not been added to the filter. If all values have been set to 1, we know either that the filter contains the element or that the bits set by its other elements have collided with the bits representing the element being queried, causing a false positive. The probability of this situation occurring can be approximated as follows:

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}, \tag{2.1}$$

where n is the number of elements in the filter [5]. Thus, as the filter approaches saturation, false positives become increasingly likely. This can be mitigated by increasing m or using multiple filters.

2.3 Cryptographic Hashing

One-way hash functions were first described by Ralph Merkle in his PhD thesis on public key cryptography in 1979 [6]. Merkle demonstrates that a hash function can serve as an efficient mechanism for checking the value of a large data field using a much smaller field, provided that it can take an input of arbitrary size and produce a fixed size output, and that there is no easy way to generate other large fields that map to the same smaller value. He observes that a suitable

function for this purpose can be defined in terms of existing cryptographic functions by encrypting a block of the large data field to obtain a fixed-size output, then combining the result with the next data block and repeating this process until the entire field is used (adding padding where necessary if there is not enough data left to make an entire block). Given a collision-resistant encryption function, Merkle proved that this process would produce a collision-resistant one-way hash of fixed size. Merkle and Ivan Damgård later derived independent proofs of the construction's security [7], [8].

Their method, known as Merkle-Damgård construction forms the basis for the MD5, SHA1 and SHA2 hash functions. From a forensic perspective, the properties of being collision-resistant and taking inputs of arbitrary size are significant in that they allow comparison of arbitrary data objects.

2.4 Use of Hashes in Forensics

A traditional forensic application of cryptographic hashes is to verify data integrity by comparing successive hashes of data objects taken over time. One benefit of this practice is the ability to track down unauthorized changes to a file system. Furthermore, this comparison strategy can be easily adapted to detect duplicate data—all objects with identical hashes can be treated as identical with an arbitrarily small probability of error. This property proves especially useful for detecting the presence of a known member in a given data set. By storing hashes of all the elements in the set and comparing these with the hash of the known object, one can quickly determine whether the set contains the object. In forensics, this provides a useful tool for locating targeted content in large collections of data, and is commonly used by examiners looking for contraband. Conversely, hashes of innocuous material such as system files can be used as a filter to identify and quickly discard content that is known to be irrelevant. To this end, efforts have been made to create databases of common files (the National Software Reference Library is perhaps the best known example [9]).

Conceptually, this approach treats the entire data object as one feature, and the hash of that feature represents the object's signature. Despite many advantages, this technique is brittle in that it is restricted to exact duplicates only: even a single bit change in either of the data objects will produce completely different hashes. Because the data objects are mapped to hash values (representing equivalence classes) uniformly at random, no relationship between the objects will be discovered. Although this "avalanche" property is desirable for many applications, it defeats any efforts to obtain more detailed information about similarities between the two objects.

2.5 Piecewise Hashing

First developed by Nicholas Harbour for dcfldd [10], piecewise hashing provides more granular comparison capabilities by subdividing the data objects and applying hashes to each subdivision so partial matches will still be detected even if some portions of the original are changed. Sector hashing, described in Young et al. [11], represents an enhanced variation on this approach that is especially well-suited for forensic drive analysis.

These schemes can be effective in many scenarios. For example, using sector hashing, a 512-byte fragment is often enough to detect the presence of a targeted file—even if other sections of it have been deleted—as long as it can be matched against a database of known sectors. Random sampling of sectors enables rapid scanning of storage media to determine with a high degree of confidence whether an object is present. Some limitations exist, however. Piecewise hashing will not detect matches in cases where the target data has been heavily fragmented or rearranged if the fragment size is smaller than the size of the data slices to which the hashes have been applied. Furthermore, if an insertion or deletion in the data causes the boundaries of the hashed pieces to become misaligned, none of the hashes will match. Young et al. observe that this is not a concern for hard drive forensics because modern operating systems align the beginning of files with disk sector boundaries [11]. This does not account for the case in which data is inserted into the beginning of a target file, however. More problematic is that no such alignment exists in network captures unless the streams are reconstructed.

2.6 Rabin Fingerprinting

Introduced in Michael Rabin's seminal paper in 1981 [12], Rabin fingerprinting presents a powerful alternative to cryptographic hashes that is capable of identifying approximate matches, Rabin's algorithm survives both fine-grained fragmentation and insertion/deletion operations through use of a sliding window that passes over the data object and makes a decision about where the hashed data should begin and end based on the properties of the data in the window. (More specifically, the algorithm views the data as a binary string and treats each bit as a coefficient in a polynomial. It then divides by a randomly chosen prime polynomial, and compares the remainder to a constant to determine its anchor points.) This method creates a set of hashed "features" that remain relatively constant in the face of many types of transformation, especially shifts in alignment of the data. This set serves as a signature (referred to by Rabin as a fingerprint) which, much like piecewise hashes, can be compared with other signatures to provide granular feedback in the form of a count of matching and non-matching features.

Two major contributions of Rabin's strategy are especially relevant to later developments in approximate matching: a context-sensitive feature-selection algorithm and the use of an efficient rolling hash. The primary advantage of the former in terms of resilience to alignment shifts has already been mentioned. It is worth noting also that an effect of this technique is more easily recognizable correlation between features and high-level content, in that data objects containing the same sequences will produce matching features more often than in piecewise hashing schemes where the content is arbitrarily divided.

In regard to the latter innovation, the sliding window that Rabin proposes is only fast to compute because it can be updated incrementally. That is, if the window has length of w bytes and l is the number of bytes in the file, the polynomial division calculation must be repeated $l - w$ times in order to compute the remainder of all possible windows. This can be achieved because the computation has been designed in such a way that the value of each window (after the first) can be calculated recursively based on the value of the previous window, with very few additional operations required.

One draw-back of Rabin fingerprinting is that there is no way to determine whether a feature is tied to a useful segment of data or not. That is, since features are selected based on arbitrary mathematical properties of the bit stream, some may correspond to interesting content while others may correspond to non-identifying attributes of the object, and there is no way of distinguishing between these two scenarios. As a result, matching based on Rabin fingerprints can produce a high ratio of false positives, which need to be eliminated through manual inspection.

2.7 Context Triggered Piecewise Hashing

In 1999, Andrew Tridgell developed the rsync algorithm [13] to provide an efficient method of synchronizing data over low-bandwidth links. The checksum employed by this algorithm leverages a scheme very similar to Rabin's in which piecewise hashing is combined with a cheaply computed, context-sensitive rolling hash. Tridgell later adapted this concept for use in his spamsum algorithm [14], which compares the resulting signatures using a weighted edit distance to detect similarity between spam emails for the purposes of blocking new versions of identified spam. In 2006, Kornblum [10] applied the spamsum algorithm to forensics in his proof-of-concept tool, *ssdeep*. Like Rabin fingerprinting, this approach involves a rolling hash function that can be updated recursively as it slides through the file. Signatures are generated as follows:

- A rolling hash is calculated for the current window of contiguous bytes.
- The resulting value is modded with a predetermined "block size" value.
- If the outcome of the mod operation is equal to one less than the block size, a "trigger point" is declared and a traditional hash is computed.
- The last six bits of this hash are encoded as a base-64 character and appended to any other base-64 characters that have been obtained already.
- If the modded hash is equal to one less than twice the block size, the same process is followed but the resulting base-64 character is appended to a second section of the signature.

Similarity scores are produced using a dynamic programming algorithm that calculates a weighted edit-distance between two signatures. A significant limitation is that only fingerprints generated with identical block sizes can be compared. Kornblum attempts to mitigate this problem somewhat by generating signatures for two block sizes, and if the block size could be set arbitrarily the problem could be solved by simply choosing a standard value. Unfortunately, the block size is a function of the number of features produced during the signature generation process. If two few features are produced to create an effective comparison, the block size will be automatically halved. Practically speaking, this has the effect of tying the block size to the length of the data object; thus the algorithm has little ability to compare objects with significantly different lengths. The fact that the number of features produced by a given block size cannot be calculated in advance creates yet more difficulty, as this may result in the process being repeated several times before an acceptable signature is generated. Finally, because only the least significant six bits are used to represent each hashed segment, there is a $\frac{1}{64}$ chance that non-matching segments will be assigned the same value, thus producing a false positive for that feature.

2.8 *sdhash*

Roussev's *sdhash* utility, introduced in 2010 [1], attempts to improve on Rabin fingerprinting by reducing the number of "weak features" used in the signature, where a weak feature is one that appears in many unrelated data objects and is therefore not closely tied to the specific data object from which it was derived [15]. The purpose of weeding out such features is to minimize the false positive rate. The intuition behind the Roussev's method for accomplishing this is that statistically uncommon features are more likely to be unique to a particular data object, and are therefore better suited to identifying it. Since the likelihood of a feature appearing is not inherently evident, *sdhash* uses an entropy-based scoring system to identify unusual features. Once selected, the features are inserted into a concatenated series of Bloom filters that forms

the object's signature.

2.8.1 Feature Selection

The features used by *sdhash* are defined simply as a 64-byte segment of the target data. Much like *ssdeep* and Rabin fingerprinting, the algorithm uses a context-sensitive sliding window during the feature selection process. Rather than calculating trigger points, however, the utility analyzes each potential feature in three steps. First, it calculates the Shannon entropy of the bytes using

$$H = -\sum_{0}^{255} P(X_i) \log P(X_i), \tag{2.2}$$

where $P(X_i)$ is estimated by counting the number of times the byte i occurs in the feature and dividing by total number of bytes in the feature. We note that there is no reason to assume this estimate bears any resemblance to the empirical probability of encountering a given byte, or even the probability of its appearance in the data object as a whole; however, it is easy to see how this calculation maintains the advantages of rolling hash used by Rabin and Kornblum. Since the next potential feature will differ from the previous in at most one byte, the total estimated entropy can be calculated recursively based on the value of the previous window. Once the raw entropy has been obtained, it is normalized on a scale from 0 to 1,000 (inclusive) using

$$H_{norm} = \left\lfloor 1000 \left(\frac{H}{\log_2 W} \right) \right\rfloor. \tag{2.3}$$

As Roussev points out [15], the connection between the value of H_{norm} and the probability of encountering a feature is complex. This is expected because, as we have already noted, there is no inherent relationship between the probability of encountering a byte given a feature, and the global distribution of the features themselves. To solve this problem, Roussev has done an empirical study [15] which explores the distribution of feature entropies for various file types. Using this distribution, he creates a mapping from each possible H_{norm} score to a "precedence ranking," referred to as H_{prec}, which ranks the H_{norm} values in ascending order of commonness.

Finally, using a second 64-byte sliding window, *sdhash* computes the "popularity" of each feature by comparing its precedence rank to that of the other features in the window. The lowest

precedence rank, corresponding to the rarest feature, gets a single point. Then the window is shifted forward one byte and the process repeated so that each feature participates in 64 comparisons before falling off the end of the window. The total number of points assigned after these comparisons is called the popularity score, H_{pop}. All features with a popularity score over the *popularity threshold* (which Roussev sets to 16) are selected and added to the signature.

2.8.2 Similarity Digests

sdhash's signatures, which Roussev refers to as similarity digests (SDBFs), are comprised of a colon-delimited header section and a series of concatenated Bloom filters. The header section contains the following components (listed in order of appearance):

- SDBF's magic string ("sdbf")
- The sdbf version number.
- The number of characters in the input name.
- The input name.
- The input size in bytes.
- The hash algorithm used (SHA1).
- The size of Bloom filters in bytes (proportional to m from Equation 2.1).
- The number of independent hash functions (k from Equation 2.1).
- A mask value for determining which bits of the 5 sub-hashes generated by splitting the SHA1 hash should be used to map a feature to the 256-byte (2^{11}-bit) filter. This value is currently set to 7ff, causing the least significant 11 bits of each 32-bit sub-hash to be used [1].
- The maximum number of features allowed in a Bloom filter (160 for continuous mode or 192 for block mode. See Section 2.8.3 for a discussion of these modes).
- The number of Bloom filters.
- Either the number of features in the last Bloom filter (continuous mode) or the block size (block mode); see Section 2.8.3.

Following the header, the Bloom filters containing the inserted features make up the majority of the similarity digest (SDBF). Roussev chooses $m = 2048$, $k = 5$ and a saturation point of either 160 or 192 bytes depending on the mode. The five hash functions are each 32-bits in length, as a result of splitting the output of SHA1 evenly, though only their least significant 11 bits are used to insert features in a filter.

2.8.3 Continuous Mode and Block Mode

Depending on the size of the input, *sdhash* defaults to one of two modes. For inputs smaller than 16 MiB, the utility runs in "continuous mode," meaning that the algorithm continues to add features to a given Bloom filter until its saturation point (which Roussev sets at 160 elements) is reached. Distance between features is ignored. For inputs greater than 16 MiB, *sdhash* defaults to "block mode," which handles signature generation slightly differently. In this mode each Bloom filter corresponds to a block of input data (the default block size is 16 KiB but this can be changed using the `--block-size (-b)` option). The algorithm selects features from each block and adds them to that block's corresponding Bloom filter until either all the features have been added or the filter reaches a saturation point of 192 elements. This mode of operation has the effect of increasing the predictability of the signature size and allowing a given filter to be mapped back to the data it was created from. In exchange for these advantages, it introduces the possibility of sparse or empty Bloom filters, as well as unexamined sections of blocks, which occur if the filter for the block reaches its saturation point early on.

2.8.4 Comparison of Signatures

The total similarity score between two objects is defined as the average of the maximum similarity filter score (SF_{score}) for each filter in the SDBF of the smaller object. The SF_{score} is a measure of the number of common bits set between two Bloom filters, with allowances made for an expected amount of overlap caused by random chance. Calculating the SF_{score} for a pair of filters involves the following steps:

- Check that there are at least six features in both filters. If not, do not perform a comparison (Roussev justifies this step by explaining that he has experimentally determined that six is the minimum number of elements needed for the score to have meaning [1].)
- Count the number of bits e_{12} that the two filters have in common.
- Calculate a cutoff point, C, representing the minimum number of common bits we require between two Bloom filters in order to be confident that they do not match purely by chance (see Equation 2.5).
- If $e_{12} \leq C$, set SF_{score} to zero.
- Otherwise, if $e_{12} > C$, use Equation 2.6 to get the value of SF_{score}.

In order to find the cutoff value C, it is necessary first to calculate an expected range of matching bits between two filters. E_{min}, the lower end of the range, represents the estimated minimum number of common bits between two filters. This is equivalent to the expected number of

common bits in filters with no common elements (that is, the number of matching bits likely to be produced by hash collisions), which is expressed as [1]:

$$E_{min} = m\left(1 - p^{ks_1} - p^{ks_2} + p^{k(s_1+s_2)}\right), \tag{2.4}$$

where p is the probability of setting a single bit (i.e., $1 - \frac{1}{m}$), and s_1 and s_2 are the number of elements inserted in the two filters.

The maximum number of matching bits, E_{max}, is estimated simply by counting the number of bits set to one in each filter and choosing the smaller count. Roussev walks through these calculations in detail in Section 3.4 of [1]. After finding the minimum and maximum, he defines the cutoff point, C, as follows:

$$C = \alpha(E_{max} - E_{min}) + E_{min}. \tag{2.5}$$

Any filters with fewer than C bits in common are treated as unrelated and given a score of zero. Filters with greater than C bits are assigned an SF_{score} according to the equation

$$SF_{score} = 100\left(\frac{e_{12} - C}{E_{max} - C}\right). \tag{2.6}$$

For each filter in the SDBF corresponding to the smaller of the two objects, this process is repeated for every filter in the SDBF of the larger object. The maximum of the resulting scores for each filter is averaged with the best scores for all other filters to produce the total score for a pair of digests.

2.9 FRASH

In light of increasing interest in approximate matching techniques, a question arises regarding how to evaluate their performance. Roussev undertook an evaluation of *ssdeep* and *sdhash* in 2011 [3] in which he carries out a number of controlled experiments, as well as a study of data from the GovDocs corpus [16]. The experiments he proposes include embedded object detection, single-common-block correlation, and multiple-common-block correlation. While providing persuasive evidence that *sdhash* offers superior performance to *ssdeep*, this work is

limited to the comparison of the two algorithms in question, and is not easily extended to others.

Building on Roussev's work, Breitinger attempts to develop a generic methodology and automated testing framework for comparing approximate matching algorithms [2]. His testing utility, FRASH (FRamework to test Algorithms of Similarity Hashing), implements Roussev's single common block test and a number of other experiments designed to measure an algorithm's response to fragment detection, alignment shifts and random byte changes. Although this tool is in the early stages of its development, it represents a first attempt to define standards against which new and existing approximate matching tools can be evaluated.

We return to this testing framework in more detail during our discussion of experimental methodology in Chapter 3.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Experiment Design and Implementation

This chapter gives an overview of our experimental methodology. We present our testing framework, built on a modified version of the FRASH framework for testing approximate matching algorithms. We also enumerate the parameters we selected for testing and their various roles in the *sdhash* algorithm. Finally, we describe the experiments we used to measure the effect of each parameter over a range of values.

3.1 Experimental Framework

Our experimental framework consists of the following components:

- a set of data files
- a customized version of *sdhash* 3.1
- a customized version of FRASH 1.01
- a Python program for parsing and graphing modified FRASH output
- a simple bash script for executing tests in parallel

Further details regarding each component are provided below.

For each experiment, we prepare a collection of parameter values. The modified version of *sdhash* allows these values to be set as command-line arguments. The commands for calling *sdhash* with each value in the collection are added to the customized FRASH framework, which treats them as separate algorithms, thus simulating the effect of compiling a separate version of *sdhash* for each parameter setting.

FRASH runs its suite of tests on *sdhash* using each parameter value in the collection. With the exception of the single-common-block test, which generates files randomly, the tests iterate through the set of data files provided, comparing each with increasingly modified versions of itself, according to the transformations defined by the test. The Python program then parses the output and produces a series of line and bar graphs. The bash script runs FRASH on all parameter collections as separate processes in parallel.

File Type	DOC	GIF	HTML	JPG	PDF	PPT	TXT	XLS
Count	10	1	23	5	28	7	8	3

Table 3.1: File Types in the t5-subset

3.1.1 Data Files

The data files used are a subset of the t5-corpus [3], which is itself a subset of the GovDocs corpus (available at http://digitalcorpora.org/corp/nps/files/govdocs1/). The t5-subset was selected by Breitinger in [2], and made available on the author's website. Breitinger uses the whole t5-corpus of 4,457 files to obtain his results for efficiency tests, and the subset for his precision and recall tests, which are much more time-consuming. Since precision and recall tests are of much greater relevance to our purpose, we run all tests against the subset.

A breakdown of the file types contained in the t5-subset is given in Table 3.1. (Note that the subset provided varies slightly from the description given in [2] and contains only 85 files used for the random-noise resistance test instead of the 100 files listed there.)

3.1.2 *sdhash*

The version of *sdhash* used in our experiments has been customized to allow access to parameters that are fixed at compile time in the original version. A full description of the function of each modified parameter is given in Section 3.2. The values tested are explained in the experiments section. No other modifications to *sdhash* have been made.

3.1.3 FRASH

FRASH offers a framework for evaluating approximate matching algorithms. It accomplishes this with five tests: efficiency, single-common-block correlation, alignment robustness and random-noise resistance. Breitinger has given a comprehensive explanation of the FRASH framework [2]. However, in order to capture the full range of behavior we observed in our versions of *sdhash*, as well to allow for more consistent metrics across all tests, we have made several significant alterations to many of the tests. The descriptions provided below correspond to out altered versions. Note that many of these alterations have since been incorporated into the FRASH 2.0 release, but unfortunately due to time constraints we were unable to integrate this new release into our testing setup. Furthermore, to our knowledge, no complete description of this new release is yet available.

In addition to the changes listed below, we have made several modifications consistently across

all tests:

- The end condition of all tests has been altered to prevent the test from terminating upon the appearance of the first zero score. Where necessary, additional end conditions have been added. These are listed in the test descriptions.
- The table values have been modified to list the average score in the data rows of every test. This required modification of the output from the single-common-block and random-noise resistance tests.
- The matching algorithm template has been altered slightly to permit initializing the object with different command line options.
- The output has been modified to include the relevant command-line arguments in order to distinguish between different parameterizations of the algorithm.
- A command-line option has been added to specify a set of algorithms to test, and to create separate temporary folders for each set so the algorithm can be run in parallel.

In general, the modifications we made to the original tests served one of two purposes:

1. to cause all tests to report their results in terms of average score, and
2. to eliminate test end-conditions that depend on the output of the algorithms being tested.

The first of these goals affected only the single common block and random-noise resistance tests, which were initially designed to report the average block size or average number of changes, respectively, that corresponded to scores falling within ten ten-point ranges from 0–10 through 90–100. We altered this with the intent of maintaining consistency with the other tests.

With respect to the goal of setting output-independent end conditions, all tests except efficiency were modified to some extent. In the original design, these tests were set to terminate when the algorithms being tested produced a score of zero, at which point no further scores were collected until the next input file. For the fragment detection and alignment robustness tests this caused the tests to terminate early and to record only non-zero scores. For the single common block and random-noise resistance tests, it was the only end condition. While this approach works reasonably well for well-made algorithms such as *sdhash* or *ssdeep*, we found it failed when run against some of our variations of *sdhash* that never produced a zero score. Furthermore, we consider it advantageous to use end conditions that do not depend on the output of the algorithms being tested, because this both prevents the algorithms from breaking or manipulating the tests with unexpected output and permits us to record output of zero. It is particularly helpful in the

case that the algorithm's scores do not decline monotonically over the course of the test. The ability to continue testing even after a score of zero has been reported also allows us to verify that this value is not an anomaly, and to describe trends in the output with greater confidence.

Efficiency

The efficiency test is divided into two parts. The first measures the speed with with the algorithm creates and compares signatures from the original data. The second gives the average length and compression ratio of the resulting signature (i.e., the ratio of signature size to file size). In the original, SHA1 is used as a baseline for comparison, and a multithreaded version of *sdhash* is included in the test of signature generation speed; these have been removed because our focus is only on the contrast between different parameterizations.

Single Common Block Correlation

Inspired by Roussev's experiment designed to evaluate the performance of *sdhash* and *ssdeep* [3], this test compares two randomly generated files containing a single common block. The comparison is performed as follows:

- Initially, the block size is set to one half the file size. On subsequent iterations, it is reduced by $\frac{1}{32}$nd of the original file size until its size reaches zero.
- This process is repeated five times, producing five scores for each block size, which are then averaged.
- As in the original version, we run the whole test three times, using file sizes 512 KB, 2,048 KB and 8,192 KB.

We have modified the end condition of this test such that it stops only when the block size reaches zero.

Random-noise Resistance

After copying its input file, the random-noise resistance test performs a series of one-byte edits on the copy. The types of edits made are deletions, insertions and substitutions, each chosen randomly with equal probability and made at randomly selected locations in the file. The insertions and substitutions both use randomly generated bytes.

We perform this test twice to show the algorithm's behavior on a large and small scale. To give the large scale view, we run the test using a step size given by

$$step = \gamma n, \tag{3.1}$$

where $\gamma = \frac{1}{100}$, n is the number of bytes in the original file, and $step$ is the number of edits performed between each score measurement. This is repeated until the number of total edits is equal to $\frac{n}{10}$. For the smaller scale we use $\gamma = \frac{1}{1000}$ and repeat until the total number of edits is equal to $\frac{n}{100}$. The reason for this approach is that, due to the randomness involved, it is difficult to determine precisely how much of the original file has been changed.

We can, however, estimate approximately how many changes would be required to ensure a score of zero. Since *sdhash* uses 64-byte features, the number of non-overlapping features in a file is at most $\frac{s}{64}$, where s is the file size in bytes. This is also the probability that a random edit will alter a given feature. Leveraging Equation 2.1, the probability of altering k features after i edits is given by:

$$\left(1 - \left(1 - \frac{1}{\frac{s}{64}}\right)^i\right) \frac{s}{64} = k. \tag{3.2}$$

Setting k equal to 99% of the total number of possible non-overlapping features gives

$$\left(1 - \left(1 - \frac{1}{\frac{s}{64}}\right)^i\right) \frac{s}{64} = .99 \left(\frac{s}{64}\right), \tag{3.3}$$

and solving this for i yields

$$\frac{-2}{\log_{10}\left(1 - \frac{1}{\frac{s}{64}}\right)} = i, \tag{3.4}$$

which predicts that for a file size of 923,136 bytes (the largest file in our data set), 99% of all *possible* features will be changed after 66,423 random edits, which corresponds to a little over seven hundredths of the number of bytes. For a file size of 4,027 bytes, the smallest in our data set, only about 125 edits are needed, about three hundredths of the total bytes.

Fragment Detection

The fragment detection test compares an input file against progressively smaller subsections of itself. The subsections are produced by repeatedly slicing off 5% of the original file size until only 5% of the file remains, then slicing off 1% of the original size until 1% remains. The slices are taken either from the end of the file, or from alternating sides, depending on the mode specified (we include results from both modes). Scores from all inputs (the set of data files) are averaged in each percentile category.

To avoid confusion, we note that this simulation differs in two important respects from common fragment-detection scenarios. First, it does not address the problem of detecting a file that may have been split into multiple fragments on a disk or in a network stream. Second, the "fragment" is analyzed as a standalone file, rather than an element embedded in other data. This latter point in particular causes difficulty for *sdhash*, as it will not process files below a minimum size of 512 bytes. Regardless, the test provides valuable information as to how the algorithms handle very similar objects of differing sizes.

Alignment Robustness

This test begins with two identical files and repeatedly transforms the second by prepending randomly generated byte sequences. Comparisons are made after each transformation. The size of the prepended block is either fixed at a specified number of bytes or determined as a percentage of the size of the input file. In both cases, we begin with the framework's original settings. Thus for the fixed-size tests we add 1 KiB blocks 4 times, for a total insertion of 4 KiB, then change to blocks of 4 KiB and continue to add these until 64 KiB has been prepended. For the percentage-based tests we set the block size to 10% and repeat the insertion until the file size doubles. Subsequently, we change to a block size of 100%, which we prepend five times.

Based on preliminary analysis of the output from these settings, we add several more variations to the fixed-size version of this test to include step sizes of 256 bytes, 64 bytes and 61 bytes, all repeated until at least 30 KiB have been inserted. These settings allowed us to see trends in the output more clearly and improve confidence that these patterns were not merely a biproduct of our sampling rate.

3.2 Parameters

The parameters selected for exploration are the *entropy rank* table (`ENTR64_RANKS`), the *popularity window* (`pop_win_size`), the *popularity threshold* (`threshold`) and the *sd score scale*

Parameter	Description	Tested Range
`ENTR64_RANKS`	Array of 1,001 integers ranging from 0–1000 inclusive; used to map calculated H_{norm} values to H_{prec} values.	See Table 3.3
`pop_win_size`	The number of contiguous features whose H_{prec} values are compared with each other in the sliding window that determines their final H_{pop} scores.	1–512
`threshold`	The minimum H_{pop} score required for feature selection; features with H_{pop} scores equal to or greater than this threshold will be inserted in a Bloom filter and included as part of the object's signature.	8–80
`SD_SCORE_SCALE`	Scaling factor for determining the minimum number of bits that two Bloom filters must have in common before they will be treated as similar (see α in Equation 2.5).	0–1

Table 3.2: Parameter names, descriptions and values tested

`SD_SCORE_SCALE`). These were made accessible by the addition of the `--entr64-ranks-index (-E)`, `--feature-threshold (-F)`, `--pop-win-p (-P)`, and `--score-scale (-S)` options, respectively. To give a sense of their roles with respect to each other and the operation of the algorithm as a whole, descriptions provided in the subsections below are presented in order of the parameter's use in *sdhash*, assuming a typical process of first generating then comparing signatures.

3.2.1 Entropy Rank Table

sdhash's feature selection process is based on H_{norm}, an estimated entropy of the potential feature (normalized to a value between 1 and 1,000), which is then mapped to an empirically determined "precedence rank" score. The score corresponds to the observed frequency of the H_{norm} score as determined by Roussev in his 2009 study on the use of statistically improbable features for finding similarity [15]. The goal is to use this mapping to find unusual features, in the hopes that these are more likely to uniquely identify a data object.

Roussev uses an array of 1,001 integers called ENTR64_RANKS to store the precedence rank values. Each of the indices in the array represents a possible H_{norm} score. The value stored at a given index is the H_{prec} assigned to that score. A low precedence rank indicates an unusual entropy, and hence a potentially distinguishing feature, which is therefore more likely to be selected.

Reference	Command Line Invocation	Null Value Range	Description
ENT_0	sdhash	[0–99], [991–1000]	The *entropy rank* table created by Roussev, unmodified.
ENT_1	sdhash -E 1	[0–99], [991–1000]	All non-null values replaced with an incrementing series from 100–990, inclusive.
ENT_2	sdhash -E 2	[0–99], [991–1000]	All non-null values replaced with an decrementing series from 900–10, inclusive.
ENT_3	sdhash -E 3	[0–99], [991–1000]	All non-null values from ENT_0 inverted (i.e., the value at the highest non-null index is switched with the lowest, the second highest with the second lowest, etc.)
ENT_4	sdhash -E 4	[0–99], [991–1000]	All non-null values replaced with randomly chosen values from 100–990, inclusive.
ENT_5	sdhash -E 5	[0–199], [991–1000]	Values from ENT_0 preserved for all non-null values.
ENT_6	sdhash -E 6	[0–99], [901–1000]	Values from ENT_0 preserved for all non-null values.
ENT_7	sdhash -E 7	[0–199], [901–1000]	Values from ENT_0 preserved for all non-null values.
ENT_8	sdhash -E 8	[0–9], [901–1000]	Inversion of ENT_1, including location of nulls. Non-null values decrement from 990–100, inclusive.

Table 3.3: Descriptions of the various *entropy rank* tables used as values for the ENTR64_RANKS parameter. The null value range indicates inclusive ranges of the table indices that were assigned null values, causing features with H_{norm} scores corresponding to these indices to be discarded.

An important characteristic of this array is that normalized entropies of 100 or lower and those higher than 991 are assigned a null score which prevents their selection (this should not be confused with a real zero value, which does not occur but would theoretically represent the most unusual feature). These values are associated with very high false positive rates, on account of there being blocks of repeated characters (on the low end) or commonly used tables (on the high end). Notably, Roussev leaves open the questions of how best to determine how many and which entropies get nulls, how to assign precedence rankings, and which data to base them on.

3.2.2 Popularity Window

During the feature selection process potential features are given a popularity score which is incremented every time it has the lowest precedence ranking (i.e., is the most unusual according to its estimated entropy) within a sliding window of potential features. In his 2009 paper on improbable features and approximate matching, Roussev represents this parameter as W [15]. In the *sdhash* code, it is assigned to `pop_win_size` and given a value of `64`, meaning that each potential feature competes with the surrounding 126 features (63 on either side) for popularity. Because an increase in the *popularity window* increases the number of contiguous features that must compete for the lowest H_{prec}, this is expected to cause higher scores to become concentrated in fewer features, which will be spread further apart from each other throughout the file. Predicting the exact impact of altered window sizes on the signatures generated from a given data set is not trivial. Section 3.2.5 describes a number of additional factors that must be considered.

3.2.3 Popularity Threshold

After the final popularity scores have been assigned to every potential feature, all features with scores above a hard-coded threshold are selected and inserted into the Bloom filters comprising the object's similarity digest. This cutoff value is stored in the threshold attribute of the sdbf_conf object in *sdhash*. Roussev has explored the effect of varying its value (which he labels t) on the algorithm's ability to find common features in similar data [15]. Although his focus is mostly on the trade off between compression and feature retention, an interesting implication of his results is that an increased threshold lowers the contrast between data containing low and high ratios of common material.

3.2.4 Similarity Digest Score Scale

As described in Section 2.8.4, two SDBFs are assigned a score equal to the average of the highest SF_{score} that can be found by comparing each Bloom filter in the SDBF of the smaller

object to every filter in the SDBF of the larger object. The SF_{score} for a given pair of filters is calculated by Equation 2.6, provided that the number of common bits between the two filters is above the cutoff given by Equation 2.5.

This cutoff value, and consequently the SF_{score} values, depend in part on the choice of α. In the source code for *sdhash*, α is represented as the constant `SD_SCORE_SCALE`, and is set to `0.3`. Roussev notes that this value has been chosen experimentally to ensure that files containing random data are given scores of zero. However, the value should also affect the sensitivity of the scores; a higher α will cause more scores to be set to zero, while at the same time creating a quicker escalation of scores for matches that fall above the cutoff.

For convenience and to reduce precision errors, we specified the `SD_SCORE_SCALE` value as an integer argument to the `--score-scale (-S)` option. This argument is divided by one hundred to obtain the actual α value.

3.2.5 Comments on Parameter Interaction

Although given a constant distribution of H_{norm} scores, we expect that increase in either *popularity threshold* or *popularity window* size will correspond to fewer features with higher H_{prec} scores, spread further apart from each other, in practice, the results vary depending on the empirical layout of the data. For example, a file in which H_{prec} scores decrease monotonically every t bytes, where t corresponds to the *popularity threshold*, could theoretically still have large groupings of hundreds of features only t bytes apart, regardless of the *popularity window* (assuming it is not so large as to prevent comparisons altogether). The maximum number of features in a grouping is equivalent to the number of distinct non-null H_{prec} values. We refer to this maximum as H_{range}. In the case that the arrangement of the H_{prec} values in the data create such groupings, the increased *popularity window* would only have the effect of increasing the distance between them. Again, this relationship is complex. The minimum distance between groupings is the sum of the *popularity window* and *popularity threshold*, but data where the H_{prec} score decreases every $t-1$ bytes would have a maximum distance between groupings of

$$W + t + (t-1)\,H_{range}. \tag{3.5}$$

An important consequence of this equation is that the distribution of selected features is not solely controlled by internal parameters. Though increasing the window size or the *popularity*

threshold may weed out weaker features, much depends on the properties of the data itself. In addition, some subtle factors, such as the choice to normalize entropy over 1000 scores, which determines the granularity of the comparison, may exert considerable influence over our results.

3.3 Experiments

Using the modified FRASH tests, we ran experiments for each of the four parameters listed, sampling values from the ranges listed in Table 3.2. We varied our method for sampling the parameter space as appropriate for each parameter, modifying or adding to the sampled values based on preliminary results.

To explore the impact of values in the *entropy rank* table, we created eight new arrays in addition to the original. The first four of these arrays preserve the positions of the one hundred null values at the beginning and ten at the end, but replace the non-null values in between. The next three arrays preserve the non-null values but vary the number of nulls at the beginning and end of the array. Finally, the last array reverses the position of the nulls and counts down from 990 to 100. Detailed descriptions of each *entropy rank* table are listed in Table 3.3.

Sampling of the remaining parameters is straightforward. For the *popularity window* we tested sizes of 16–160 in increments of 16, then 128–512 in increments of 64. We also tested values of 1 and 15 as a baseline (since these should both produce zero features). For the *popularity threshold* we tested 8 to 80 in increments of 8. Finally, we tested *sd score scale* values in increments of .1 from 0 to 1, then again in increments of .01 from 0 to .1.

Results and analysis of all four experiments are given in Chapter 4.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Experimental Results and Analysis

We divide our results into three major categories. The first contains observations regarding the efficiency tests. These were significantly more informative than we predicted and provide a useful introduction to our analysis. The second section describes general observations derived from the outcome of the precision and recall tests. These pertain to all parameters and serve as a baseline for understanding the fine grained variations that follow. In the third and final section we present results for each of the precision and recall tests across the spectrum of parameter values tested.

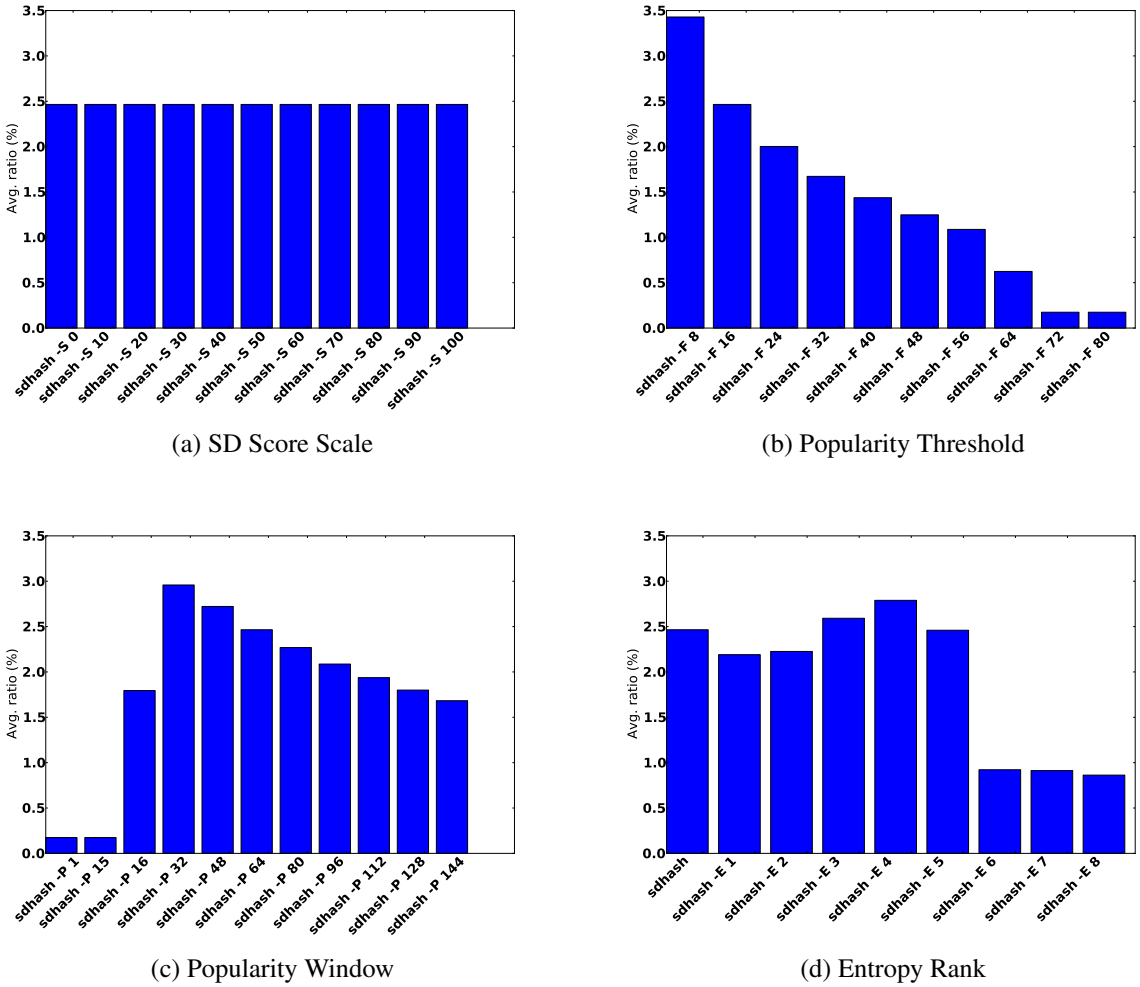
4.1 Efficiency Tests

Although *sdhash*'s space efficiency and speed are not the primary focus of this research, the results from these tests provide useful preliminary insight into the underlying effects of varying the four parameters. We hypothesized that some small change in these values might be produced by our modifications to *sdhash* as a result of fluctuations in the number of features selected. However, especially with respect to signature size, we initially underestimated the impact of this change. We examine the results here collectively to highlight broad differences and similarities across all parameters.

4.1.1 Compression Ratio

In continuous mode, the length of the non-header portion of the similarity digests that *sdhash* uses as signatures depends on the number of features the algorithm selects from the target data. From a research perspective, this is the primary advantage of running our tests using continuous mode rather than block mode, which generates a new filter for every block regardless of the number of features contained. The results of our signature compression ratio measurements (see Section 3.1.3), confirm the expected relationship between number of features and signature size. As a group, they serve as an overview of how the parameters are affecting the number of features chosen.

Of the four parameters analyzed, *sd score scale* is the only one that has no impact on the feature selection process; it is used solely for calculating comparison scores between already-existing digests. As expected, Figure 4.1a demonstrates that the signature size remains constant across



(a) SD Score Scale

(b) Popularity Threshold

(c) Popularity Window

(d) Entropy Rank

Figure 4.1: The average ratio of signature size to file size for *sdhash* signatures shown across all tested values for each of the four parameters investigated.

all tested values. The average compression rate of 2.46% matches that of *sdhash* using the default parameter settings, as shown in the first bar of Figure 4.1d where the original *sdhash entropy rank* table is used, and again in Figures 4.1b and 4.1c when the *popularity threshold* and *popularity window* match Roussev's choices of 16 and 64, respectively.

Increasing the *popularity threshold* causes a clear decline in signature size. Again, this is in line with expectations since higher thresholds mean fewer features will be chosen. To appreciate the implications of Figure 4.1b and Figure 4.1c, however, it is necessary to consider the relationship between *popularity threshold* and *popularity window*. Each of these parameters was

varied while holding the other constant at its default value. Thus for all *popularity thresholds*, the maximum possible H_{pop} was 64. `sdhash -F 64` selects only the features that received this maximum score. Threshold values higher than 64 (72 and 80) are unattainable, and the signatures produced using these settings have minimum sizes (corresponding to space required to store header information and one empty Bloom filter). This explains why the ratio for the last two values of the *popularity threshold* compression tests are identical to the first two values of the *popularity window* compression test, which also choose no features. Although the ratio declines somewhat steadily, the exact distribution of H_{pop} scores is an empirical property of the data files used. Further work is needed to demonstrate that this set is representative.

Changes in the size of the *popularity window* show a similar trend, with the number of features selected generally decreasing as the window increases. The relationship is slightly more complicated, however. As mentioned above, the first two bars, corresponding to window sizes of 1 and 15 (both smaller than the default *popularity threshold*) represent an essentially empty signature. This is followed by an initial low in the chart at window size 16. Because the window size and default *popularity threshold* are equal, this setting chooses only features that received a maximum score by having the lowest precedence rank during every comparison over the sliding window. The situation is analogous to `sdhash -F 64` in the previous figure, though in this case many more features are selected because a perfect score of 16 out of 16 is easier to obtain than a perfect score of 64 out of 64.

After reaching a peak at window size 32, the ratio decreases gradually. Figure 4.1c shows the shape of this decrease out to a *popularity window* size of 160. Sampling at increments of 64 from window size 128 to 512 shows a continuation of this trend. Unlike Figure 4.1b, there is no easily defined cutoff point beyond which no features are chosen. On the contrary, at least one feature (having the leftmost global minimum H_{pop}) should be selected unless the window size is set so close to the file size that there is not sufficient space to perform the number of comparisons needed to exceed the popularity threshold, or the H_{prec} values are distributed in such a way that no single feature passes the threshold before the window hits the end of the file. Again, extensive empirical study is required to determine what entropy patterns, if any, are typical.

As described in Section 3.2.1, the *entropy rank* tables differ from the other parameters in that they are a set of distinct matrices, rather than a single scalar tested over a range of values. For this reason we do not expect to see obvious trends in Figure 4.1d. Still, several important

relationships stand out: foremost, *entropy rank* tables six, seven and eight all have very similar compression ratios (0.92%, 0.91% and 0.86% of file size, respectively), and are much lower than the others. This suggests that they share a common characteristic that causes them to pick many fewer features. Because the only alteration made to table six was to increase the number of high H_{norm} scores that receive a null rank (and are therefore never selected), and this same alternation was also made in tables seven and eight, we argue that it is the overriding factor. This implies that the features *sdhash* chooses from the data files tend to be clustered toward the very top of the possible H_{norm} rankings. Removing these from the selection pool results in much small signatures.

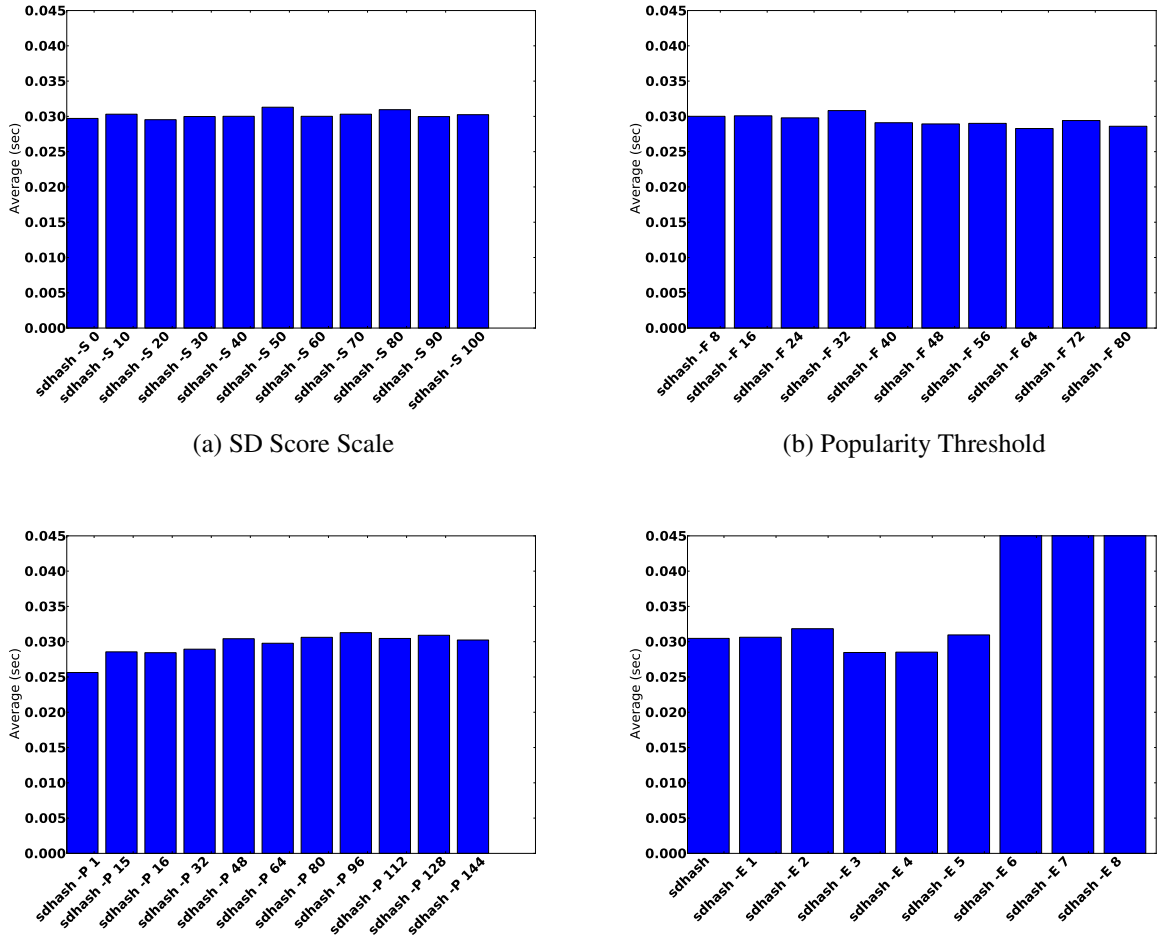
Corroborating this conclusion is the fact that ENT_5 has an average compression ratio of 2.46%, less than 0.01% below the compression ratio of *sdhash* using Roussev's settings. Although it is possible that the different tables are causing different features to be selected—an effect that would not be detectable from signature size alone—evidence presented in later sections will continue to support the theory that the main difference between the two parameters is that running *sdhash* with table 5 has dropped a relatively small number of low-entropy features.

Leaving the null values in places and replacing the original scores with either an incrementing or decrementing series (as in ENT_1 and ENT_2) corresponds to a small decrease in signature size. Further work with a larger sample would help to substantiate this claim, but intuitively it makes sense: a feature is selected when it stands out from those around it. Since its H_{norm} score is very likely to be close to that of its neighbors, a ranking table that assigns similar rankings to adjacent H_{norm} scores should create a smooth curve of H_{prec} scores that has fewer decisive minimums than a table where adjacent H_{norm} scores can "jump" to significantly different ranks. Likewise, it is not surprising that ENT_4, in which the nulls were left in place but the remaining ranks were assigned at random, chooses more features on average.

Finally, there appears to be a loose correlation between tables in which the values have been inverted (without changing the position of the nulls). The two pairs of tables with this inverse relationship (1 and 2, as well as 0 and 3), have similar compression ratios, and exhibit similar behavior in several other tests.

4.1.2 Signature Generation Speed

As noted in the previous section, the reason the *sd score scale* parameter has no effect on the compression ratio is that it is used only during signature comparison. It follows that this



(a) SD Score Scale

(b) Popularity Threshold

(c) Popularity Window

(d) Entropy Rank

Figure 4.2: The average time in seconds required to generate signatures, shown across all values of each of the four parameters investigated.

parameter should also have no affect on signature generation speed. Thus, although there is some slight variation in speed shown by Figure 4.2a, this can be safely attributed to noise in the system, since there is no plausible reason why processing speed should be affected by a parameter that is not invoked during the signature generation process. This observation is chiefly useful as a gauge for reading the other generation speed graphs, where we can also assume that small variations are to be discarded. For this reason,the only notable speed difference is in the last three tested versions of the *entropy rank* table.

If we are correct in assuming that a large fraction of the features have H_{norm} scores above 900,

which causes these three tables to assign them H_{prec} scores of null, a possible explanation for the additional time spent creating signatures is that the null value is simultaneously treated as a special (non-numeric) indicator and as a zero. This causes problems with one of *sdhash's* speed optimizations, which allows the *popularity window* to continue to slide right as long as three conditions are met:

1. The rightmost byte in the new window must not have an H_{prec} score less than the current minimum,
2. The index of the current minimum must not have yet reached the end of the window, and
3. The window must not extend past the end of the data object.

While these three conditions are true the window can proceed linearly through the data. If one of the conditions is not satisfied, however, *sdhash* will attempt to search the *popularity window* for a new minimum. In an extreme case, this would mean $n \times p$ comparisons, where n is the input size and p is the length of the *popularity window*.

If it happens that the newest score in the list is less than the current minimum, it would be possible to immediately replace the current minimum with this score. *sdhash* does not do so, however. Rather, the algorithm searches the entire window from the beginning until it reaches the new minimum (despite the fact that this is guaranteed to be the last element in the window, i.e., the newest score). This could be considered a minor efficiency bug in *sdhash's* implementation. The dual treatment of the null value compounds the problem in two ways. First, because it is interpreted as a zero, it ensures that the first of the three conditions for inexpensively sliding the *popularity window* is violated whenever a null appears at the rightmost edge of the window. This will cause the entire window to be re-scanned but will never change the minimum H_{prec} because the null is automatically disqualified from the comparison and everything else in the window has already been compared (provided that the second condition still holds). Second, once the null proceeds to the leftmost side of the window it will be set to the current minimum in accordance with the algorithm's search procedure, which begins with the assumption that the leftmost element is a minimum and then scans for elements with which to replace it. Because it is a null, its H_{pop} score will never be incremented. Since it is treated as a zero, no elements with a lower score will be found and the algorithm will again do nothing for p comparisons until it can move on to the next value, dropping the null off the window. The combination of these factors suggests that increasing the number of nulls in the data should slow it down noticeably, as Figure 4.2 confirms.

4.2 Precision and Recall: General Results

Although the precision and recall tests provide a much more detailed view of our parameter space than the efficiency tests, a broad characterization of the results gives helpful context for interpreting the variations associated with the individual parameters. For this purpose it is sufficient to look at the behavior of Roussev's utility with its factory default settings still in place.

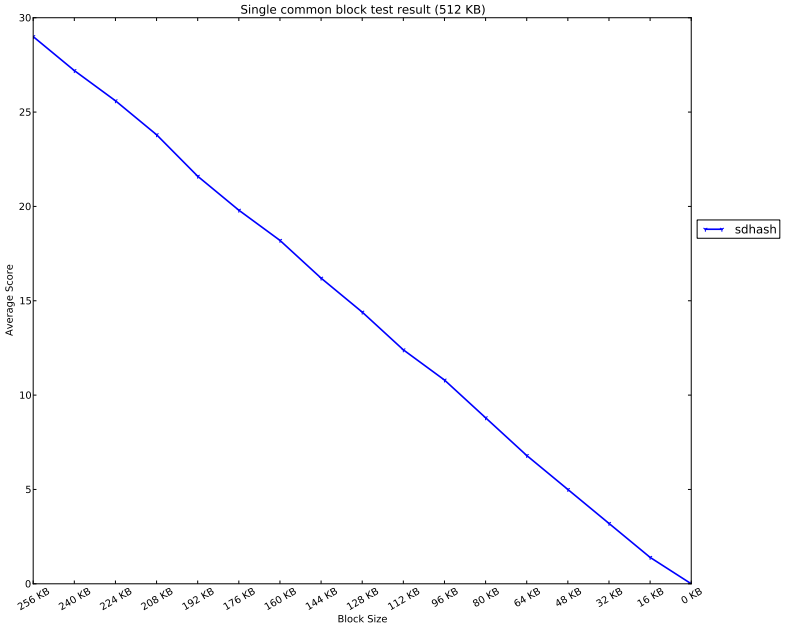
A quick visual inspection of the test results is sufficient to suggest that the tests fall into one of two categories: those in which the score diminishes in proportion to diminishing similar material, and those in which the score remains relatively constant (or oscillates within a range).

4.2.1 Commonality

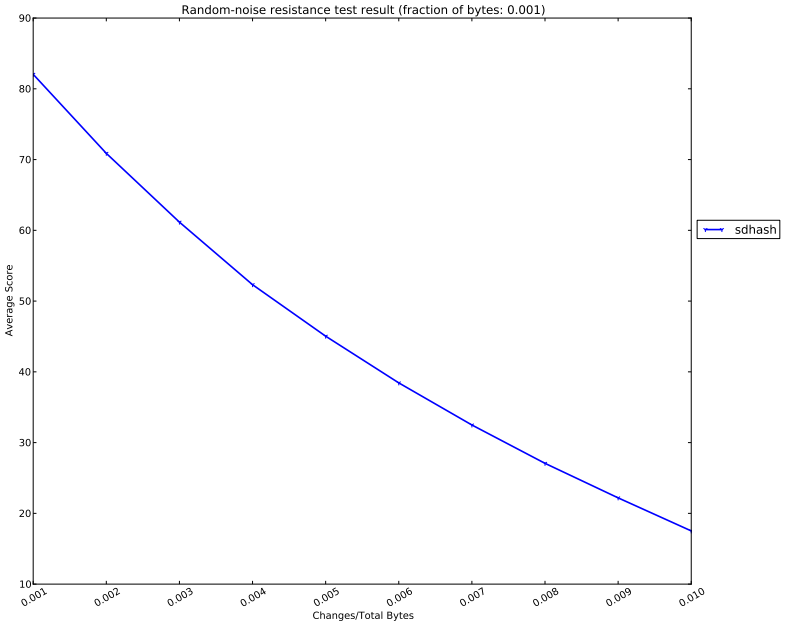
The first category includes the single common block and random-noise resistance tests. We refer to these as "commonality" tests because their results are easily correlated to a measurement of the amount of common material that *sdhash* detects between the two data objects (although in the case of the random-noise resistance the detection-rate degrades very quickly). For both tests, Figure 4.3 shows a smooth decay in the average score as the material common to both files is reduced or transformed. In the single-common-block test decay represents the diminishing size of the common block, which in turn produces a diminishing number of Bloom filters with matching features, and thus a lower average maximum SF_{score}. A similar phenomenon is occurring in the random-noise resistance test, except in this case the common material is not actually being eliminated. Rather, it is being modified at a granularity of less than 64 bytes as random transformations damage the 64-byte features that are being used to create the signatures. This demonstrates a weakness in *sdhash*'s design, causing the algorithm to fail to identify similarity even though a large amount of material remains the same.

4.2.2 Containment

The alignment and fragment detection tests fall into the second category. We call these "containment" tests because their output is most easily understood as indicating the presence of a fragment in a larger object. Figure 4.4a shows scores oscillating from the low 50s to the low 90s. This occurs because the alignment shifts are produced by random blocks of 256 bytes added to the beginning of the file. These blocks have relatively constant entropy, and experimentation suggests that they tend to produce about the same number of features (an average of 4.47 features per 256 bytes of random data). Because we are running *sdhash* in continuous

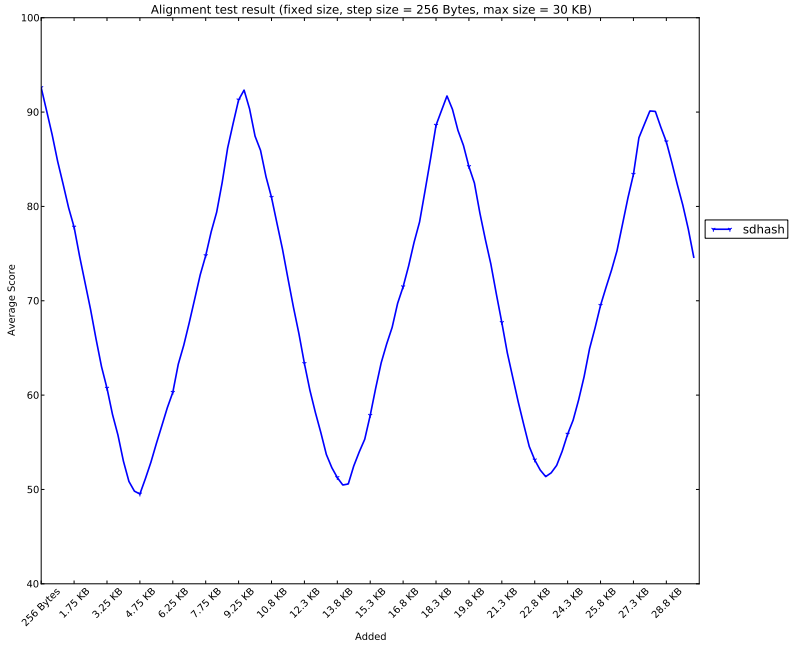


(a) Single Common Block in 512 KiB files

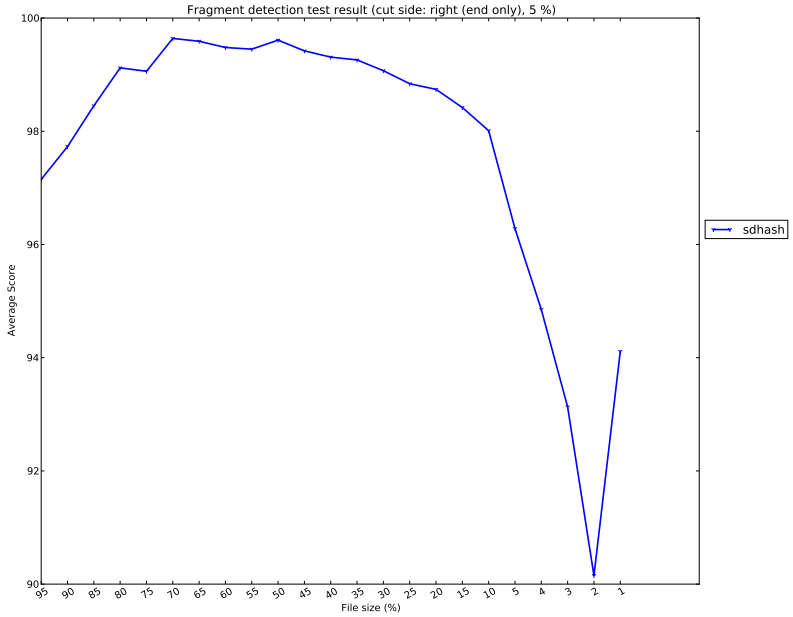


(b) Random-noise Resistance

Figure 4.3: Precision and recall tests in which *sdhash* measures "commonality." In these tests the average score diminishes as the proportion of similar material is reduced. Scores shown are for the factory default settings of *sdhash*.



(a) Alignment for 256 Byte shifts



(b) Fragment Detection

Figure 4.4: Precision and recall tests in which *sdhash* measures "containment." In these tests the average score remains constant or fluctuates within a range as long as some fragment of the smaller object can be detected in the larger. Scores shown are for the factory default settings of *sdhash*.

mode, this causes a chain-reaction in which the features added to the first bloom filter in the digest cause the same number of features to be pushed back to the next filter. As a result, all the filters will be a out of alignment and the scores will decline. We note that the first minimum of the curve occurs at around 4.75 KiB. By our estimate, this corresponds to approximately 85 features. Allowing some latitude owing to some fluctuations in the number of features produced from random data, this appears to confirm our conclusions. The maximum interference produced by an alignment shift should occur when 80 features are added. As the blocks continue to shift the filters come back into alignment and the score returns to its peak in the low 90's. This pattern continues even if the amount of random noise added eventually dwarfs the amount of original common material, clearly demonstrating that the test describes containment rather than commonality.

Similarly, Figure 4.4b shows the average score hovering in the high 90's until only 3% of the original file remains. At this point, it begins to dip, but never drops below 90, and eventually rebounds to around 94. We posit that this fluctuation at the right end of the graph occurs for the reasons similar to the fluctuation in the alignment graph. The cut in the file causes the last Bloom filter to lose features. It then receives a low match score which is averaged in with the high scores from the beginning of the file. As the overall size of the file diminishes, this low score begins to have a larger impact, causing the fluctuation to become more noticeable.

4.2.3 Relative File Size and *sdhash* Behavior

We emphasize that the two categories of tests are in fact exhibiting the same behavior under different circumstances. *sdhash*'s scores always measure the extent to which the smaller object is contained in the larger. Although the decline in scores exhibited by the commonality tests can be correlated with the reduction of common features between two compared objects, it is crucial to note that this effect is a direct product of the objects' relative sizes. Both commonality tests compare objects that are either exactly the same size or within a few bytes of each other. When the file sizes are held constant and are close to each other, the proportion of common material dominates the score. Because the amount of common material in both files is by definition the same, it is easy to see that when the files are of similar size the proportion of common material is also the same.

In contrast, when the proportion of common material in the smaller file is constant—either because common material is removed in proportion to file size, as in the fragmentation test, or because the smaller file is not changed, as in the alignment test—then so is the score, and the

test results highlight the presence of the smaller file. This indifference to unmatched material in the larger file represents a major advantage of *sdhash*, causing it to be well-suited to forensic tasks that require searching for very small targets in large amounts of data (Roussev refers to this as the "needle in a haystack" scenario [1]). Unlike other algorithms, such as *ssdeep*, it can operate successfully on arbitrarily sized inputs (provided they make the minimum size cutoff of 512 bytes).

Although the correlation between *sdhash* scores and common material in the single-common-block and random-noise resistance tests is largely an artifact of the tests' design, we can leverage this circumstance to observe the effects of our parameter settings on the algorithm's ability to indicate commonality. We will argue in Chapter 5 that analysis of these effects are generalizable and can be applied to a modified *sdhash* that allows the user to request a commonality score.

4.3 Precision and Recall: Results By Parameter

We proceed with an examination of the impact of our altered parameter settings on *sdhash*'s ability to identify similarity in each of the four tests. Results for each experiment are given roughly in order of increasing complexity of the manipulated parameters, beginning with *sd score scale*, continuing through popularity threshold and *popularity window* size, and ending with the *entropy rank* tables. This organization enables later analysis to build on earlier observations.

4.3.1 SD Score Scale

The results of our SD Score Scale experiments show a consistently linear relationship across all precision and recall tests. Increasing the value of the parameter produces a decrease in average scores in all scenarios. Though scores converge at some points (which we discuss individually), in no case did scores from a higher parameter setting cross above the scores from a lower one.

An argument of 100 (i.e., setting the parameter to 1) causes the algorithm to report a score of zero for all comparisons. This confirms expectations because at that setting Equation 2.5 reduces to $C = E_{max}$, meaning that no filters could have enough common features to produce a score.

At the other end of the range, an argument of 0 prevents *sdhash* from ever reporting a zero score. Again, this is in line with expectations: Equation 2.4 shows that two Bloom filters each containing 160 elements but having no elements in common should have an overlap of approximately 214 bits. This gives a value of 586 for the denominator of Equation 2.6, meaning a swing of only 24 bits above the expected overlap would yield a similarity score of 4. Since the maximum SF_{score} is selected, it is likely that many filters will receive an SF_{score} above zero, and the chances that the averages of all these maximums will fall to zero for the overall similarity score are very low. Roussev notes that his choice of 0.3 for this parameter was made with the intent of ensuring that this situation did not arise, and objects with no features in common are always assigned a score of zero.

Single Common Block Results

The single-common block test shows a linear reduction in average scores for all *sd score scale* values with the exception of `sdhash -S 100` which always returns zero, and `sdhash -S 90`, which returns 0 until the file size is increased to 8 MiB. This change in output is likely produced by the corresponding increase in block size, which improves the probability of encountering

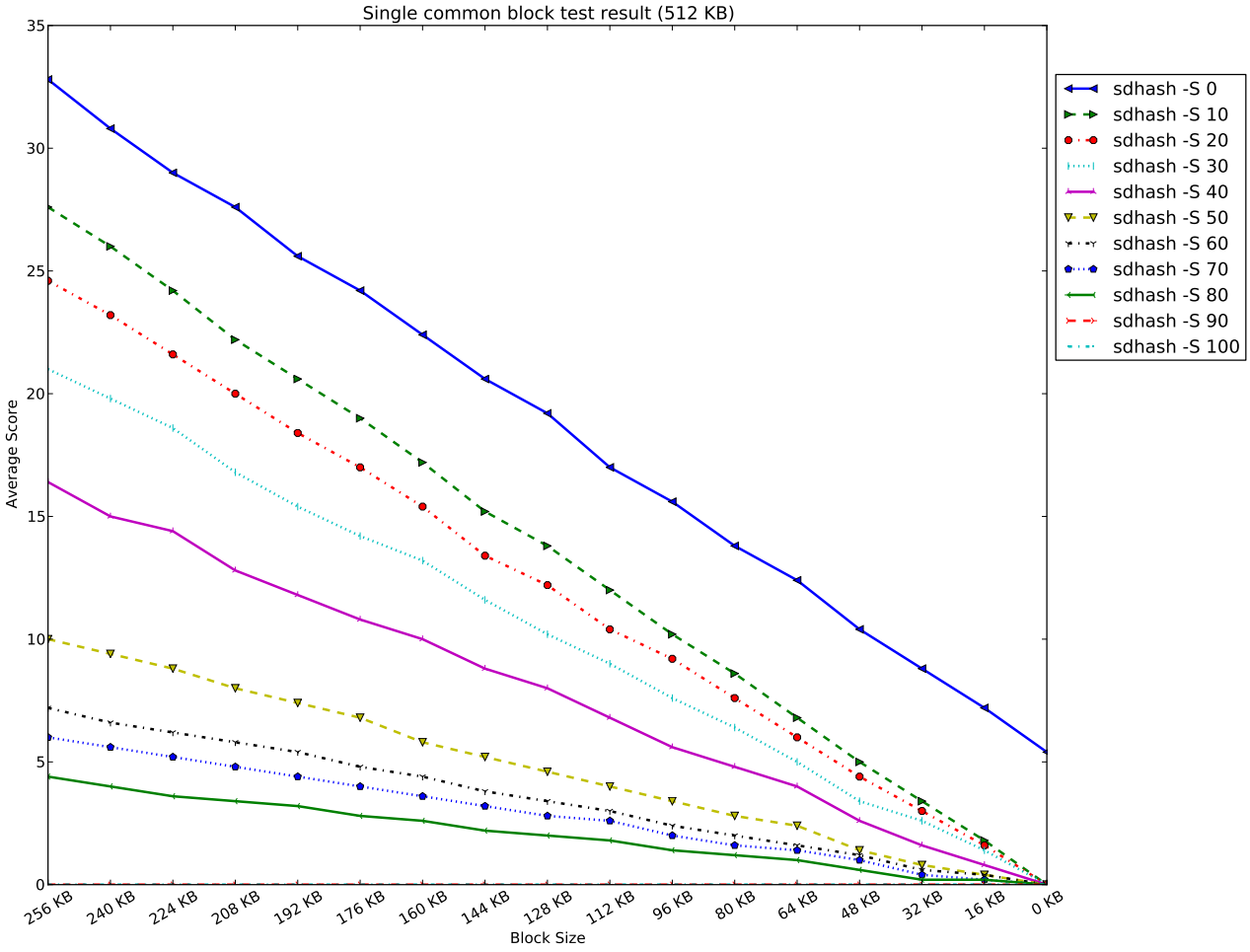


Figure 4.5: Average similarity scores for comparison of randomly generated 512-KiB files with a single common block of decreasing size, measured using *sd score scale* values between 0 and 100. `sdhash -S 30` matches Roussev's settings. `sdhash -S 90` and `sdhash -S 100` are flush with the x axis.

filters that meet the stringent matching criteria that a value of .9 demands.

For values in the range of 10–80, Figures 4.5, 4.6 and 4.7 show a direct relationship between an increase in the parameter value and a decrease in the slope of the output, with the steepest slope occurring at `sdhash -S 10`. This loss of contrast is a consequence of the fact that as C approaches E_{max} the number of filters receiving non-zero SF_{score} values reduces, causing even high-scoring filters to be muted as they are averaged with filters receiving SF_{score} values of

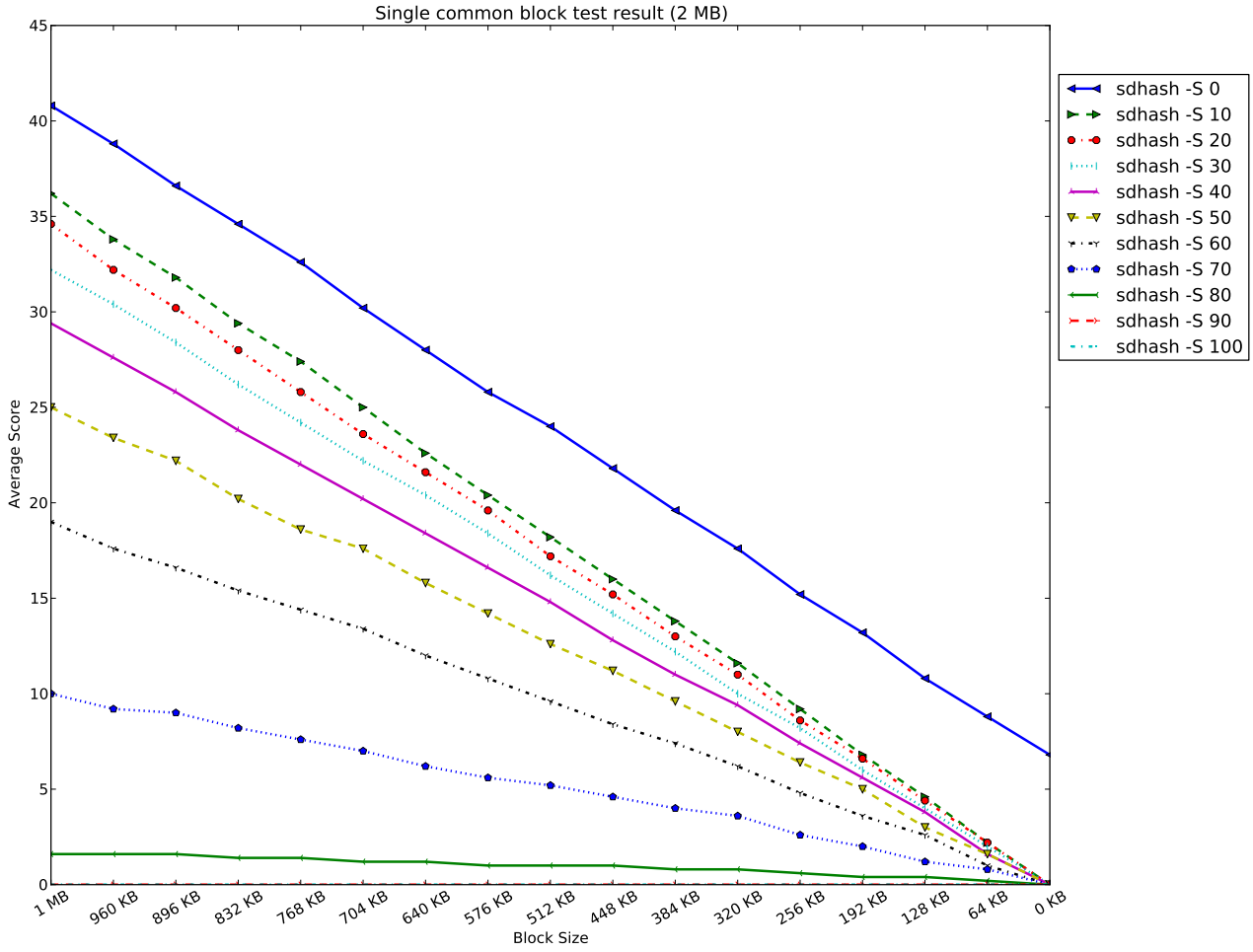


Figure 4.6: Average similarity scores for comparison of randomly generated 2-MiB files with a single common block of decreasing size, measured using *sd score scale* values between 0 and 100. `sdhash -S 30` matches Roussev's settings. `sdhash -S 90` and `sdhash -S 100` are flush with the x axis.

zero. Equivalently, higher sd score settings cause the algorithm to treat more filters as though they have nothing in common—that is, as though the common bits between are the product of random chance.

For values less than 10, the slope remains constant but the y intercept of the line increases. This represents the point where non-matching filters begin to receive scores, causing score inflation across all block sizes. Figure 4.8 shows sd score settings from ranging from 0 to .1 in increments

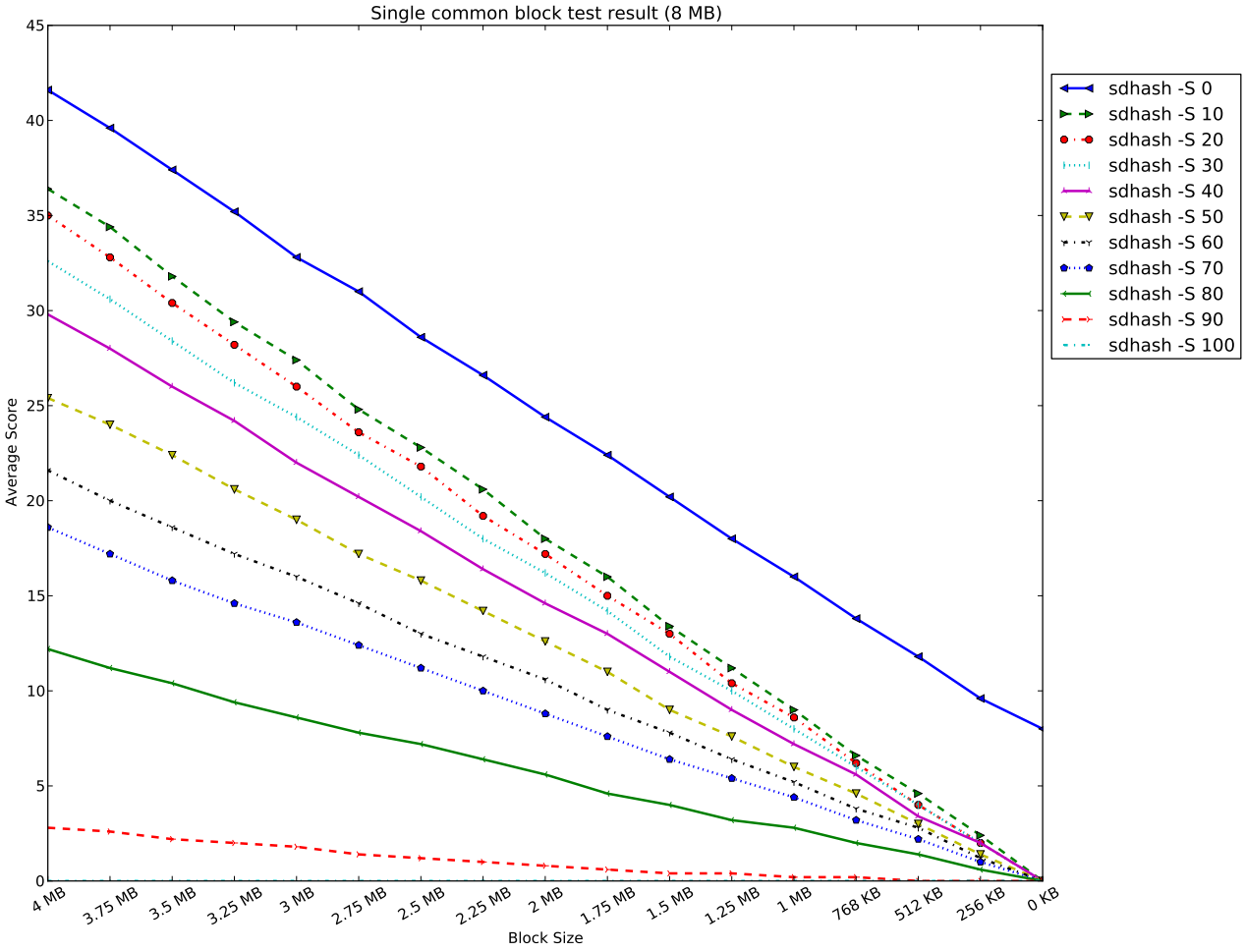


Figure 4.7: Average similarity scores for comparison of randomly generated 8-MiB files with a single common block of decreasing size, measured using *sd score scale* values between 0 and 100. `sdhash -S 30` matches Roussev's settings. `sdhash -S 100` is flush with the x axis.

of .01, illustrating this effect in greater detail.

While Roussev's choice of 0.3 for this parameter satisfies his goal of preventing dissimilar objects from receiving scores greater than zero, results from this test present a case for lowering the value to 0.1, provided that this does not introduce problems in other contexts. If the output is interpreted to indicate a measure of commonality between the two files, we assert that it is preferable to use parameter settings that show greater contrast as the amount of common material diminishes.

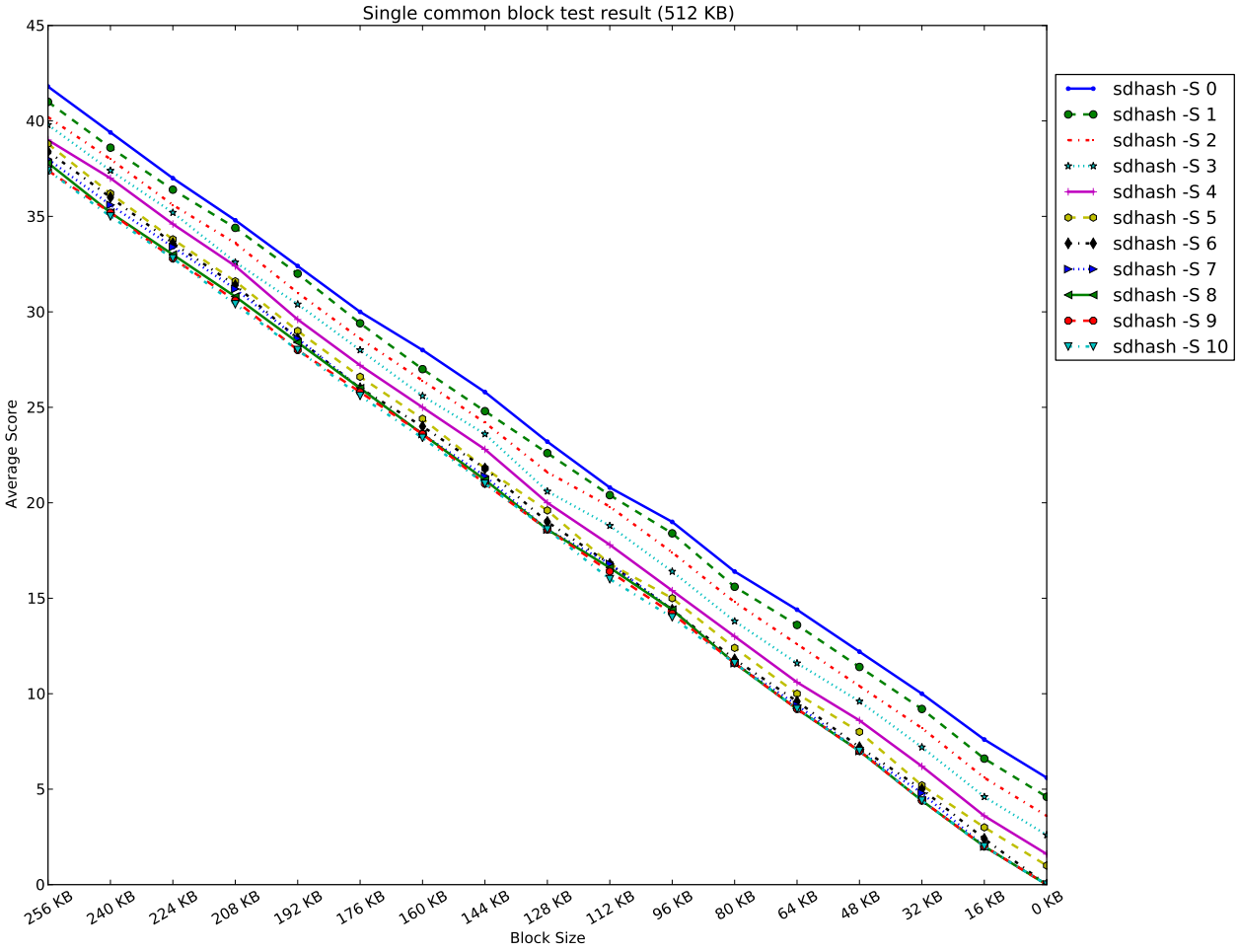


Figure 4.8: Average similarity scores for comparison of randomly generated 512-KiB files with a single common block of decreasing size, measured using *sd score scale* values between 0 and 10

Even if we adopt Roussev's similarity score threshold to change output into a binary decision as to whether similarity has been detected, a setting of 0.1 still succeeds at the smallest ratio of block-size to file size for all possible choices of a threshold. Roussev uses this same criteria to argue that *sdhash* has better detection capabilities than *ssdeep* [3].

Random-noise Resistance Test Results

Figures 4.9 and 4.9 give a large- and small-scale views, respecively, of the outcome of the random-noise resistance test. Because this test rapidly damages identifying features, it quickly renders sdhash ineffective at identifying similarity, despite the fact that a large portion of common material remains intact. Higher settings of *sd score scale* accelerate the algorithm's failure, and we can see that values above 0.5 have already dropped the average score to zero by the first

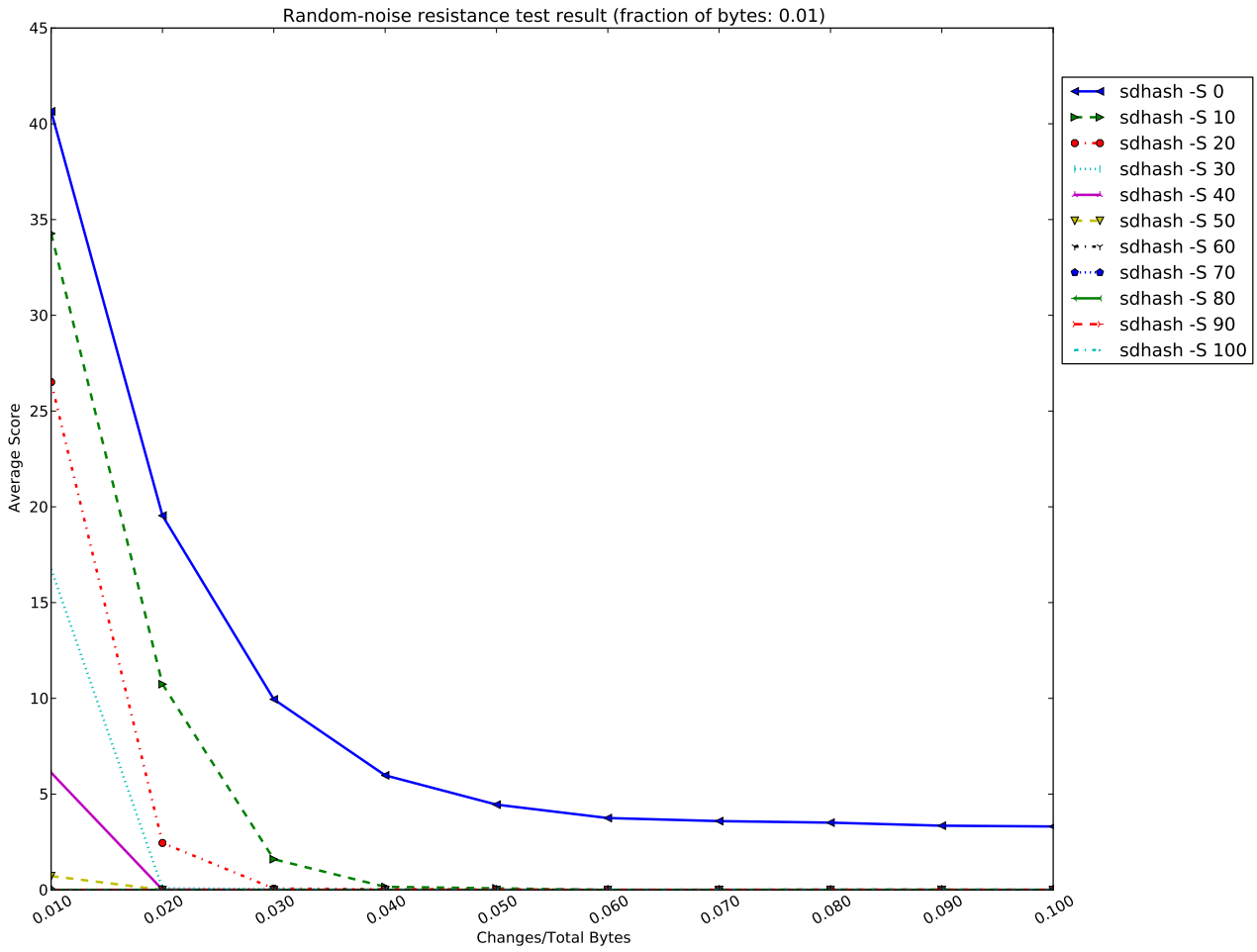


Figure 4.9: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using *sd score scale* values between 0 and 100 (number of transformations = $\frac{1}{100}$ of total bytes in original file). `sdhash -S 30` matches Roussev's settings. `sdhash -S 60` and higher are flush with the x axis.

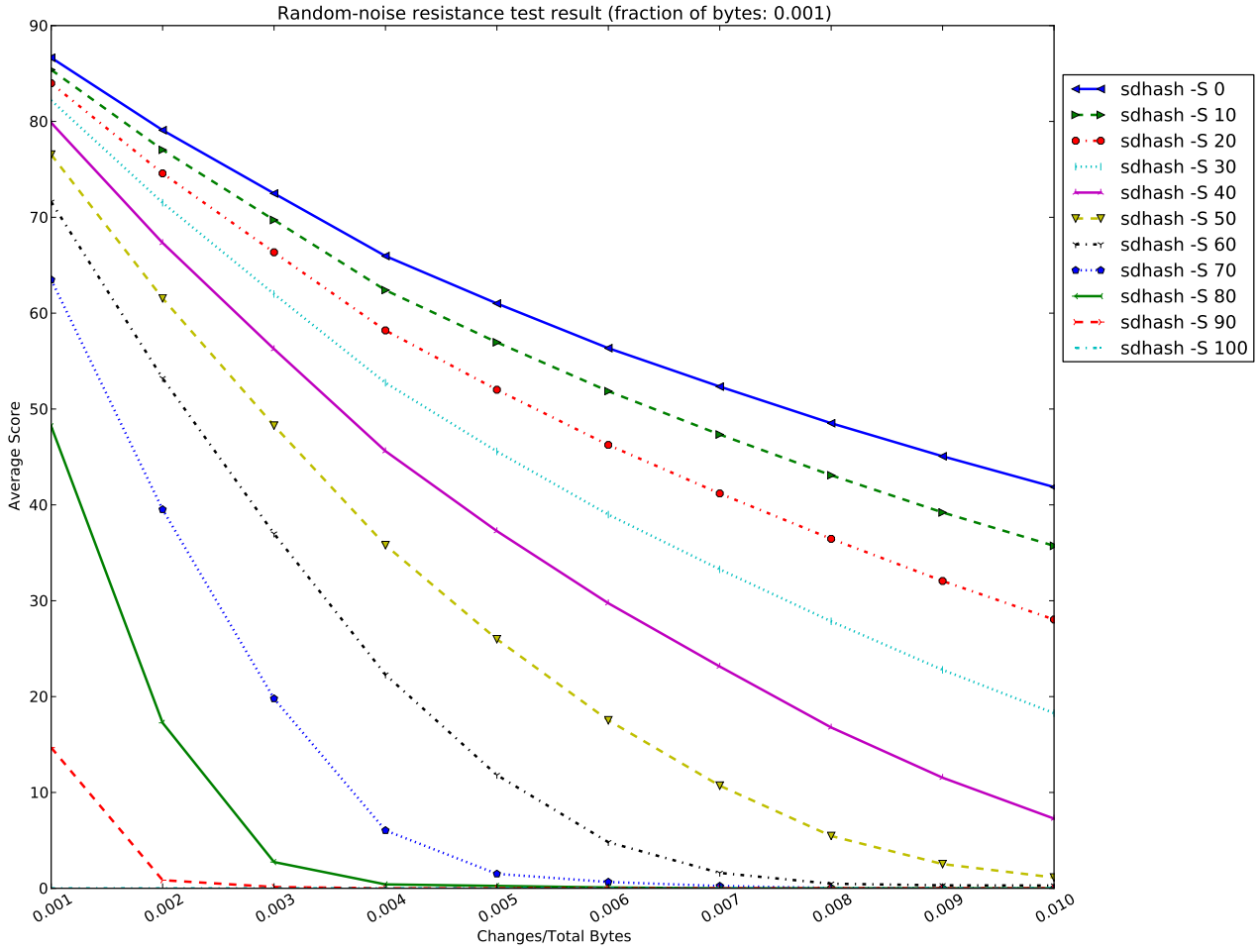


Figure 4.10: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using *sd score scale* values between 0 and 100 (number of transformations = $\frac{1}{1000}$ of total bytes in original file). `sdhash -S 30` matches Roussev's settings. `sdhash -S 100` is flush with the x axis.

measurement, which occurs after approximately one hundredth of the bytes in the file have been altered (see Section 3.1.3 for a more precise explanation of the step size).

Although at first blush, `sdhash -S 0` appears the most resistant to this attack, it is ruled out as an alternative on account of its failure to reach 0 even when the compared objects are completely unrelated. In fact, closer inspection shows the curve plateaus around a score of 5 after approxi-

mately 6% of the bytes have been changed. Comparison with the results of the single common block test reveals that this is indicative of no relationship between the compared objects.

Notably, `sdhash -S 10` again gives the best results, showing contrast across the widest range of comparisons before eventually failing. While even this parameter setting shows significant weakness for this test, it outperforms the others by a considerable margin.

Figure 4.10 shows samples at one tenth the step size of Figure 4.9, allowing us to see the output of parameterizations using *sd score scale* values of over 0.5. This confirms the trend set in the large-scale view. Scores decline most quickly at the outset, when a transformation has the largest probability of affecting an untouched feature, then taper off in a smooth curve.

Alignment Test Results

The alignment tests begin with two identical files and this kernel of similar material remains unmodified throughout the course of the transformations. All alterations take the form of dissimilar material added to the second of the two objects. Because one of the objects is constantly increasing in size, optimal behavior for this test should not show contrast as the files become less alike. Rather, we hope to see consistently high scores, indicating that one of the objects is

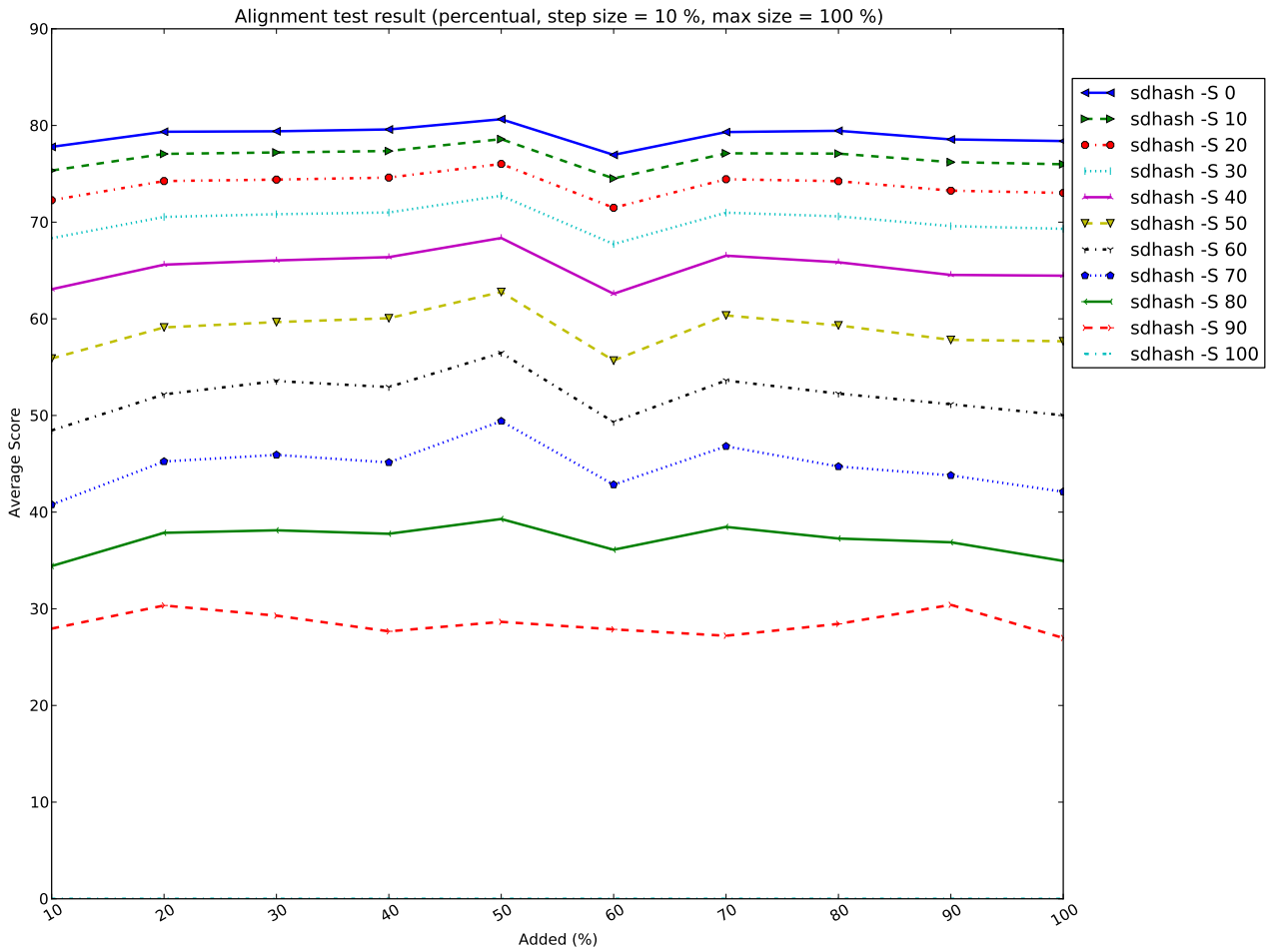


Figure 4.11: Average similarity scores for comparison of files against copies of themselves with chunks of random data prepended, measured using *sd score scale* values between 0 and 100 (chunk size = 10% of file size). `sdhash -S 30` matches Roussev's settings. `sdhash -S 100` is flush with the x axis.

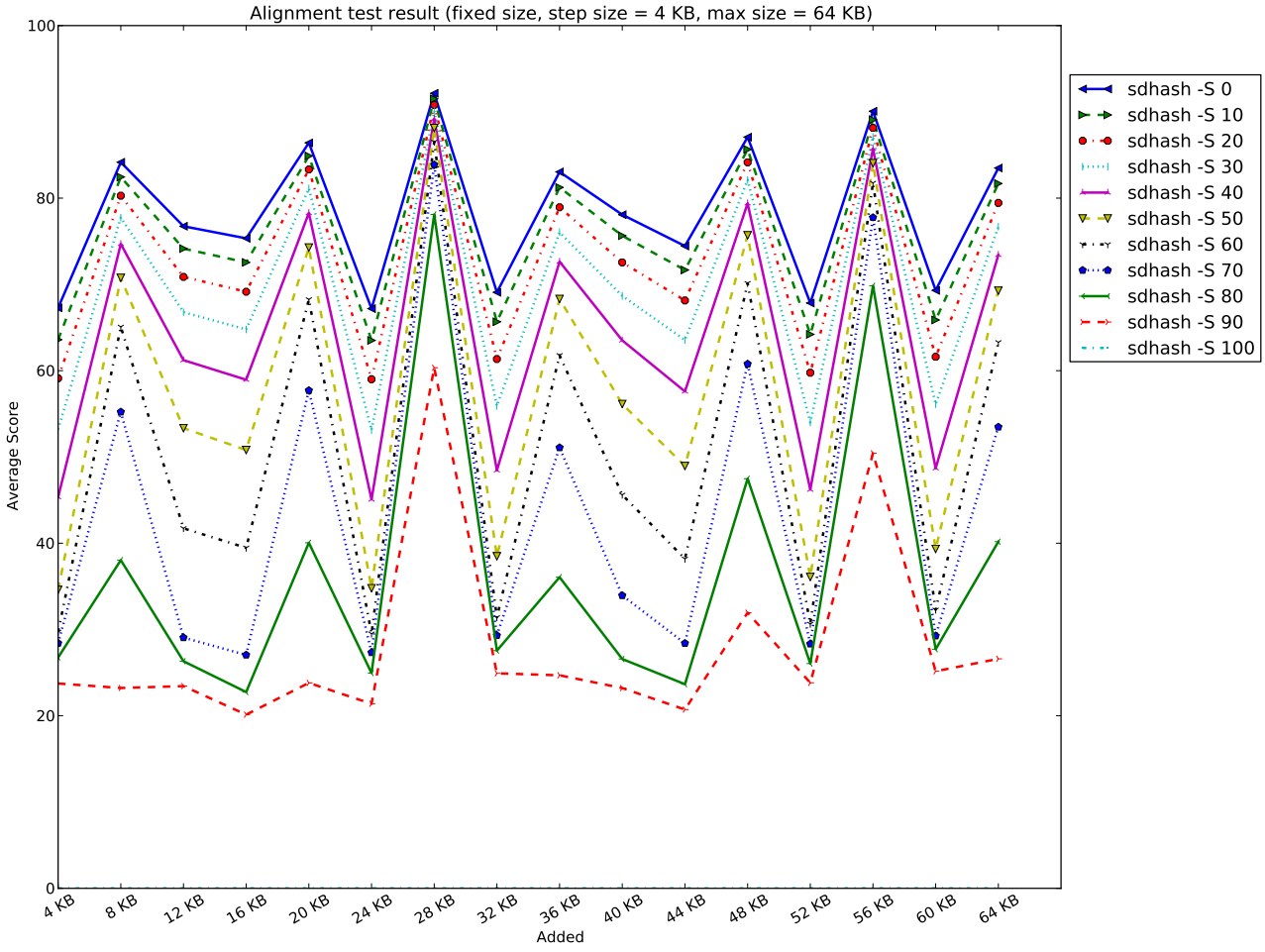


Figure 4.12: Average similarity scores for comparison of files against copies of themselves with chunks of random data prepended, measured using *sd score scale* values between 0 and 100 (chunk size = 4KiB). `sdhash -S 30` matches Roussev's settings. `sdhash -S 100` is flush with the x axis.

exactly contained in the other.

As Figure 4.11 demonstrates, *sdhash* exhibits exactly this behavior for large enough alignment shifts. The scores reported form a relatively flat horizontal line for all *sd score scale* values. The main distinguishing feature is how high of an average score the parameter tends to assign. A higher value is better here, since a exact copy of one of the objects being compared is contained in the other object. Furthermore, the flatter the line, the more consistent performance

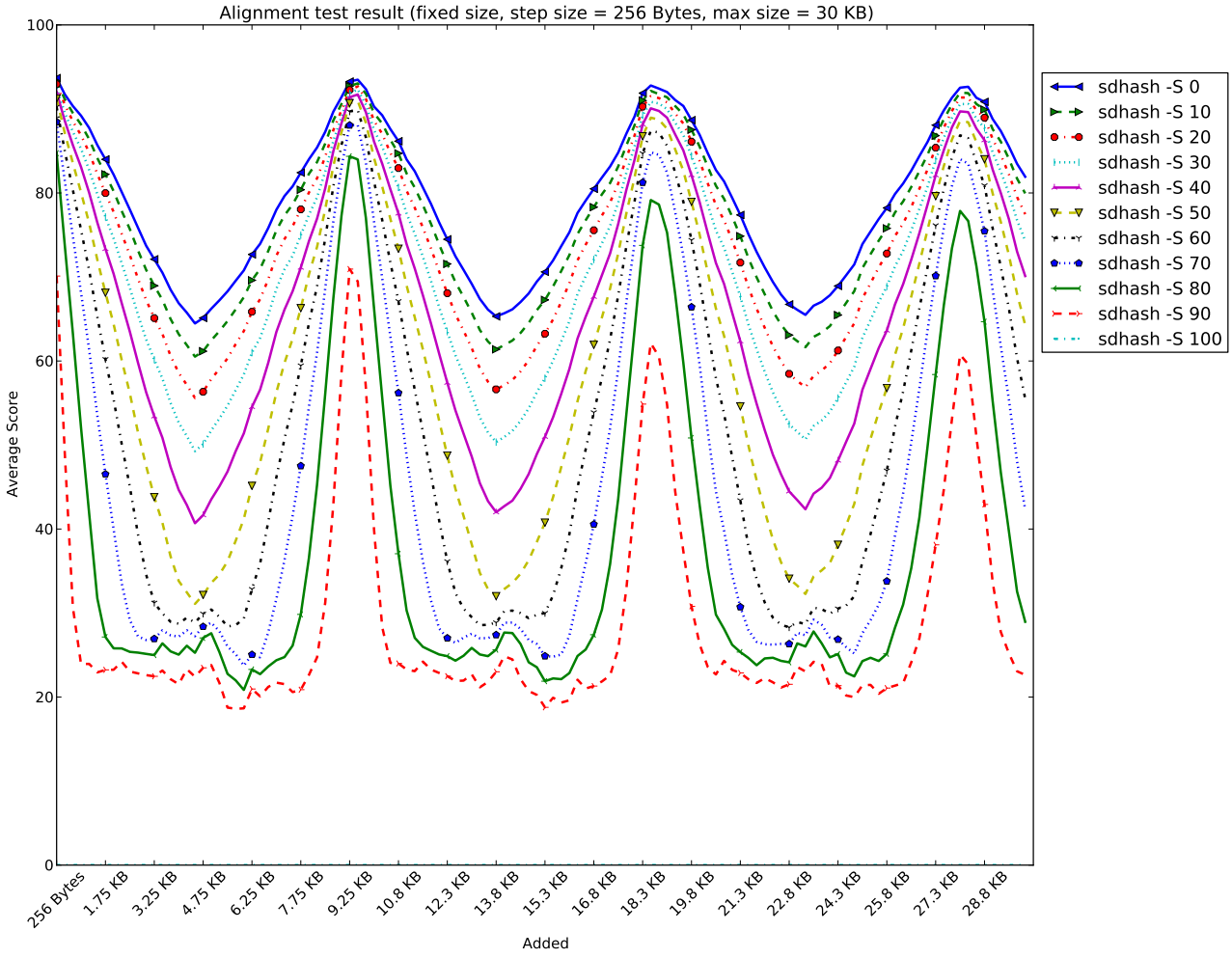


Figure 4.13: Average similarity scores for comparison of files against copies of themselves with chunks of random data prepended, measured using *sd score scale* values between 0 and 100 (chunk size = 256 bytes). `sdhash -S 30` matches Roussev's settings. `sdhash -S 100` is flush with the x axis.

demonstrated.

Since the shifts added are determined by percentage of file size and the file sizes of our data set vary significantly, this graph is showing a average of different-length shifts. This averaging has the effect of flattening the fluctuations one would predict as a result of the inserted random data displacing features at the beginning of the signature and forcing them from the end of one Bloom filter to the beginning of the next. In other words, this percentage-based mode of the test

has the unfortunate property of concealing the interference it is attempting to measure.

Switching to fixed shifts of 4 KiB removes the smoothing effect and a saw-toothed pattern emerges, as shown in Figure 4.12. The various parameterizations maintain their order, but converge toward clear peaks and valleys. In order to obtain a more detailed look, we switch to a smaller step size and take a larger number of samples, covering a total shift of 30 KiB. (As mentioned in Section 3.1.3, we also tested increments of 64 KiB and 61 KiB to confirm that the shape of the curve was not unduly influenced by our sampling rate. In this and in the alignment tests that follow in subsequent tests, the graphs produced by these alternate sample rates were identical.) This produces a much smoother curve, presented in Figure 4.13. Again, the parameterizations maintain their vertical ordering, and follow the same peaks and valleys, producing waves that have a period of approximately 9 KiB, corresponding to the amount of random data required to produce a full Bloom filter of features (i.e., 160 features in continuous mode).

The waves with the smallest amplitude represent the most consistent behavior. This property correlates directly to the size of sd score value used. Larger values result in curves with larger amplitudes. Conveniently, this means that the same settings that yield the highest scores also give the most consistent results. The best of these is `sdhash -S 0`, but `sdhash -S 10` is a close second. Again, this argues in favor of `sdhash -S 10` as an optimal parameter value.

Fragment Detection Test Results

As in the previous containment test, we expect *sdhash* to report a consistently high score in the fragmentation test. Although the similar material is reduced by the transformations of the test, so is the total size of the smaller object. As a result, the ratio of similar material to the size of the smaller object is always the same (specifically, the ratio is always one, since the fragment is completely contained in the larger object). In this case, unlike the commonality tests, optimal

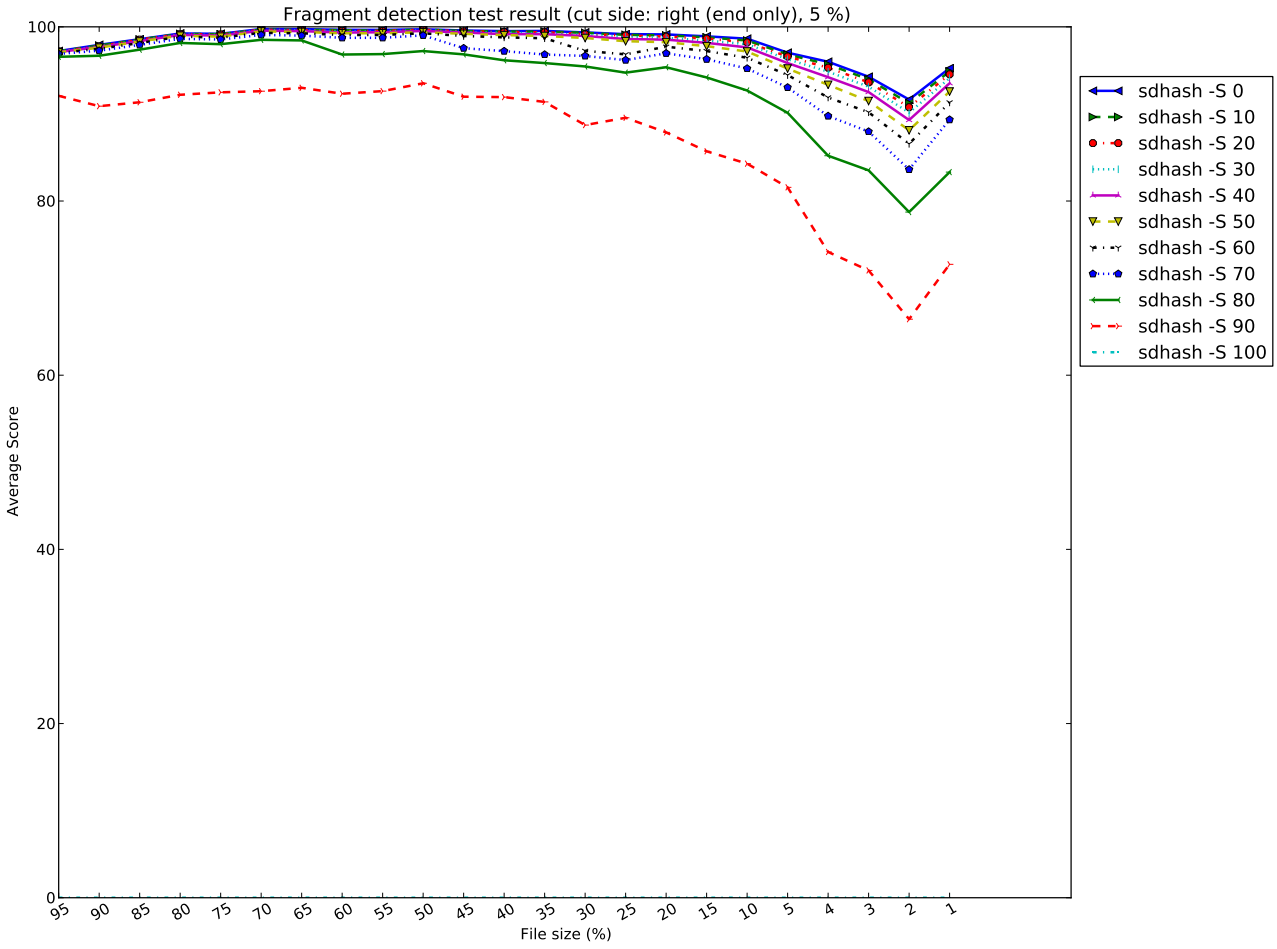


Figure 4.14: Average similarity scores for comparison of files against copies of themselves with slices removed from the tail, measured using *sd score scale* values between 0 and 100 (slice size = 5% of file size until 5% remains, then 1%). `sdhash -S 30` matches Roussev's settings. `sdhash -S 100` is flush with the x axis.

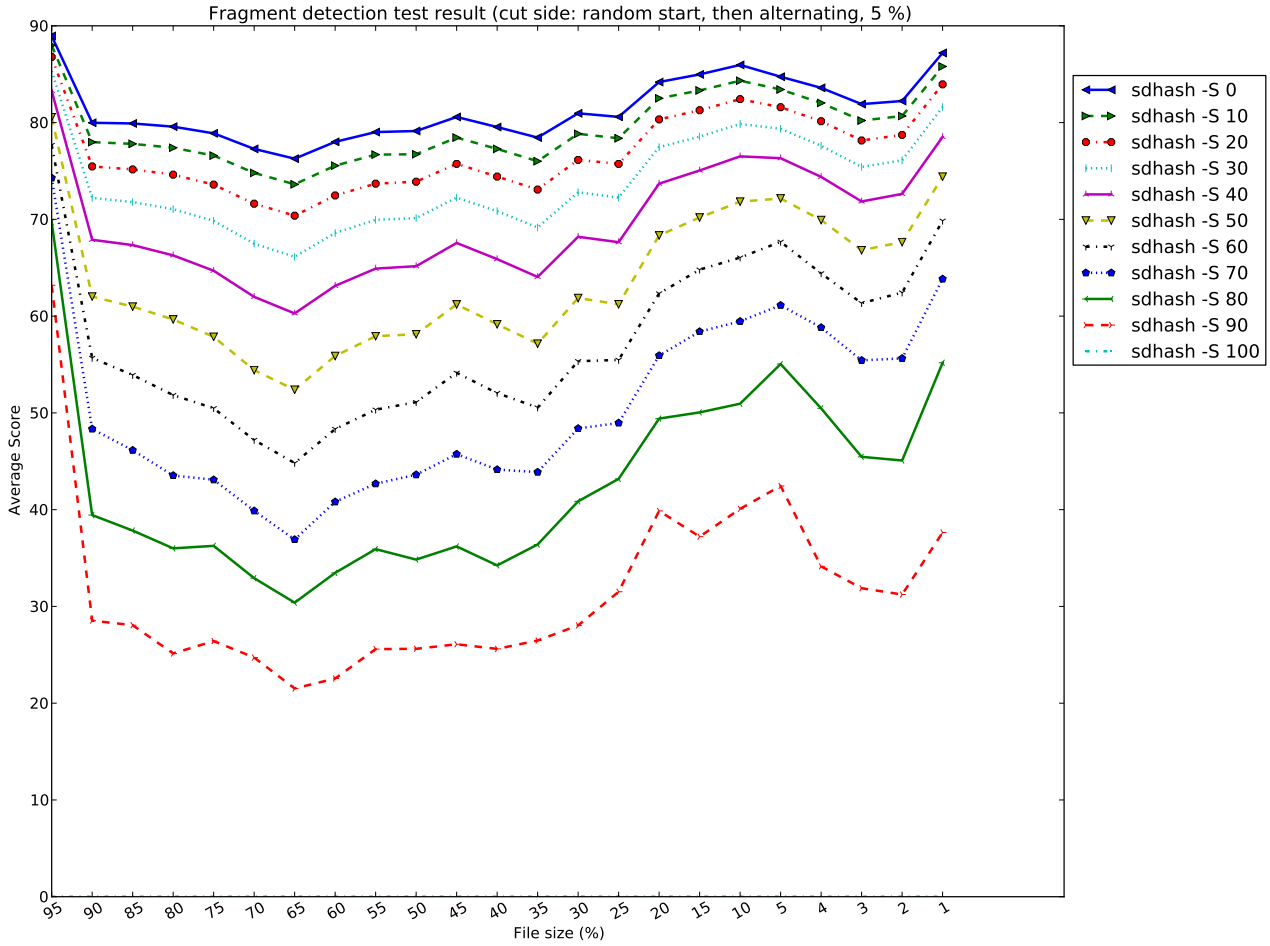


Figure 4.15: Average similarity scores for comparison of files against copies of themselves with slices removed from alternating ends, measured using *sd score scale* values between 0 and 100 (slice size = 5% of file size until 5% remains, then 1%). `sdhash -S 30` matches Roussev's settings. `sdhash -S 100` is flush with the x axis.

behavior does not show contrast as the size of the fragment decreases. Figure 4.14 illustrates that this behavior holds across all tested values except `sdhash -S 100`. Results from settings below 0.4 are nearly indistinguishable, though lower scores maintain a slight advantage. This is expected for *sdhash*, as trimming from the end causes no changes in the allocation of features to Bloom filters with the exception of the last filter in the signature, which loses features unless the cut falls exactly at the boundary between two filters.

The second fragment detection test, which produces its fragments by cutting from opposite sides at the file, creates more difficulty for *sdhash* on account of the alignment problems we have already seen in Section 4.3.1. As in Figure 4.13 from that section, Figure 4.15 demonstrates that the parameterizations with the highest score also exhibit the most consistent behavior. `sdhash -S 10` is the best choice that reliably gives a zero score when none of the targeted data is contained in the larger object.

4.3.2 Popularity Threshold

Unlike *sd score scale*, variations in the *popularity threshold* do not display a consistent linear relationship with average similarity scores. Moreover, parameter settings that perform well under some circumstances fare less well in others. While some tests demonstrate strong relationships between the chosen values and *sdhash*'s behavior, others appear to show little correlation. Especially when the tests involve random data, this may indicate that the performance of the parameters is context sensitive and that the overriding factor is the makeup of the data on which it is run.

As predicted, both `sdhash -F 72` and `sdhash -F 80` cause feature selection to be impossible and give scores of 0 for all comparisons. Going forward, we omit these from discussion.

Single Common Block Results

Each of the three runs of the single common block test using different file sizes results in a different arrangement of curves. Figures 4.16, 4.17 and 4.18 illustrate these variations. We suspect this outcome may be a consequence of the fact that files are generated randomly. Additional testing showed that variations occurred even among different runs of the same file size. Further investigation is required to determine which, if any, of the parameter settings is favored over a series of runs. In our initial observations, we see no clear argument for preferring one parameter setting over another.

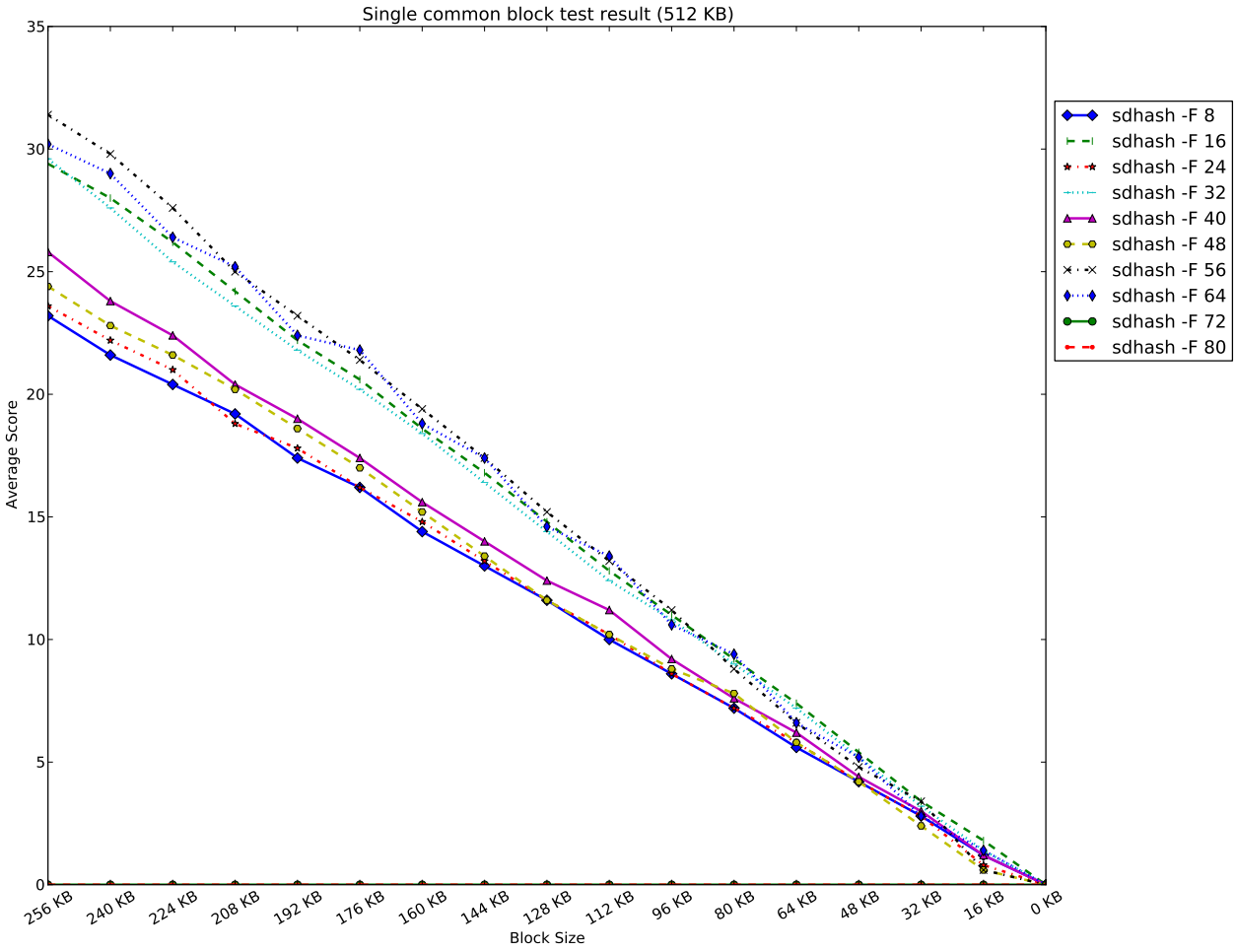


Figure 4.16: Average similarity scores for comparison of randomly generated 512-KiB files with a single common block of decreasing size, measured using *popularity threshold* values between 8 and 80. `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

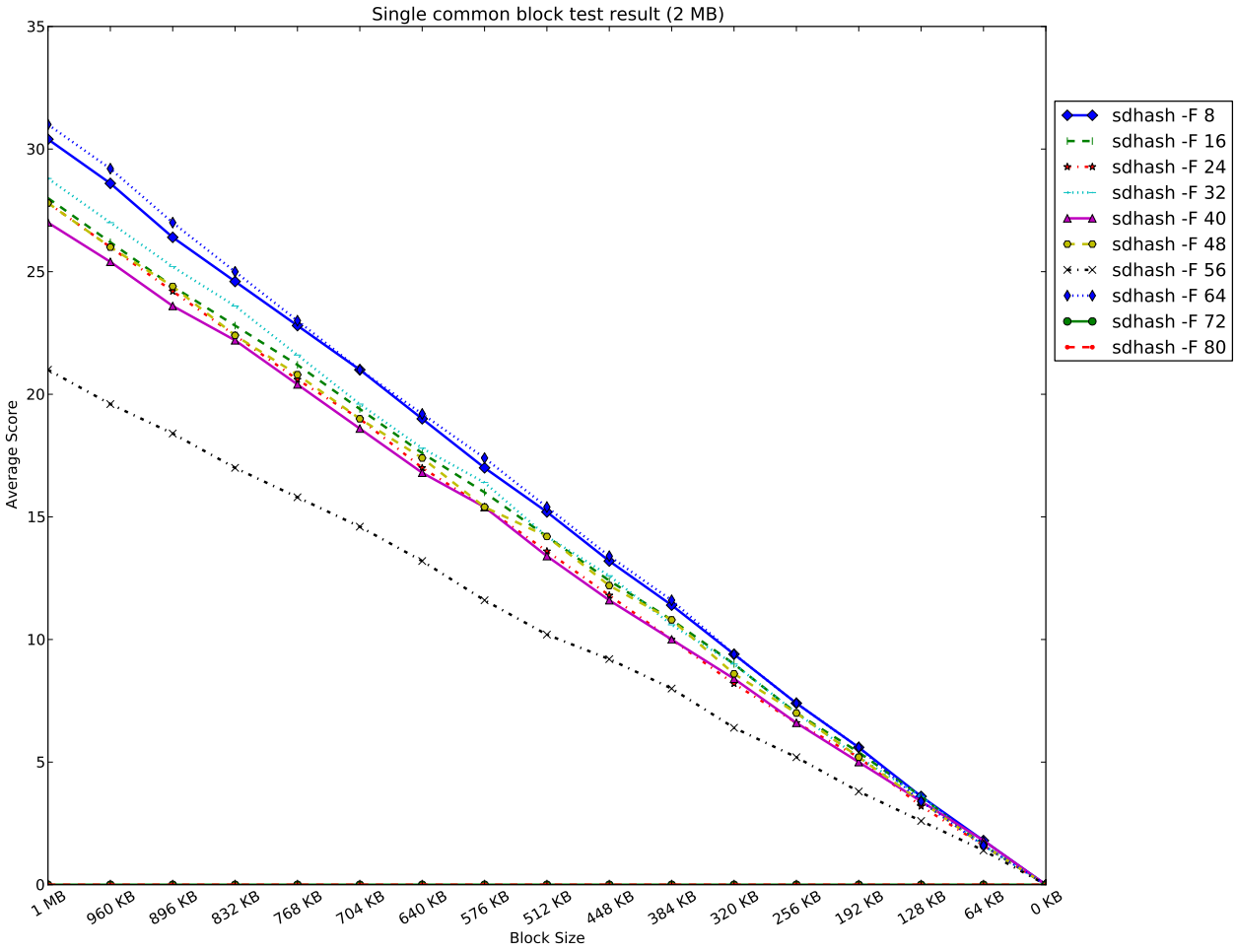


Figure 4.17: Average similarity scores for comparison of randomly generated 2-MiB files with a single common block of decreasing size, measured using *popularity threshold* values between 8 and 80. `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

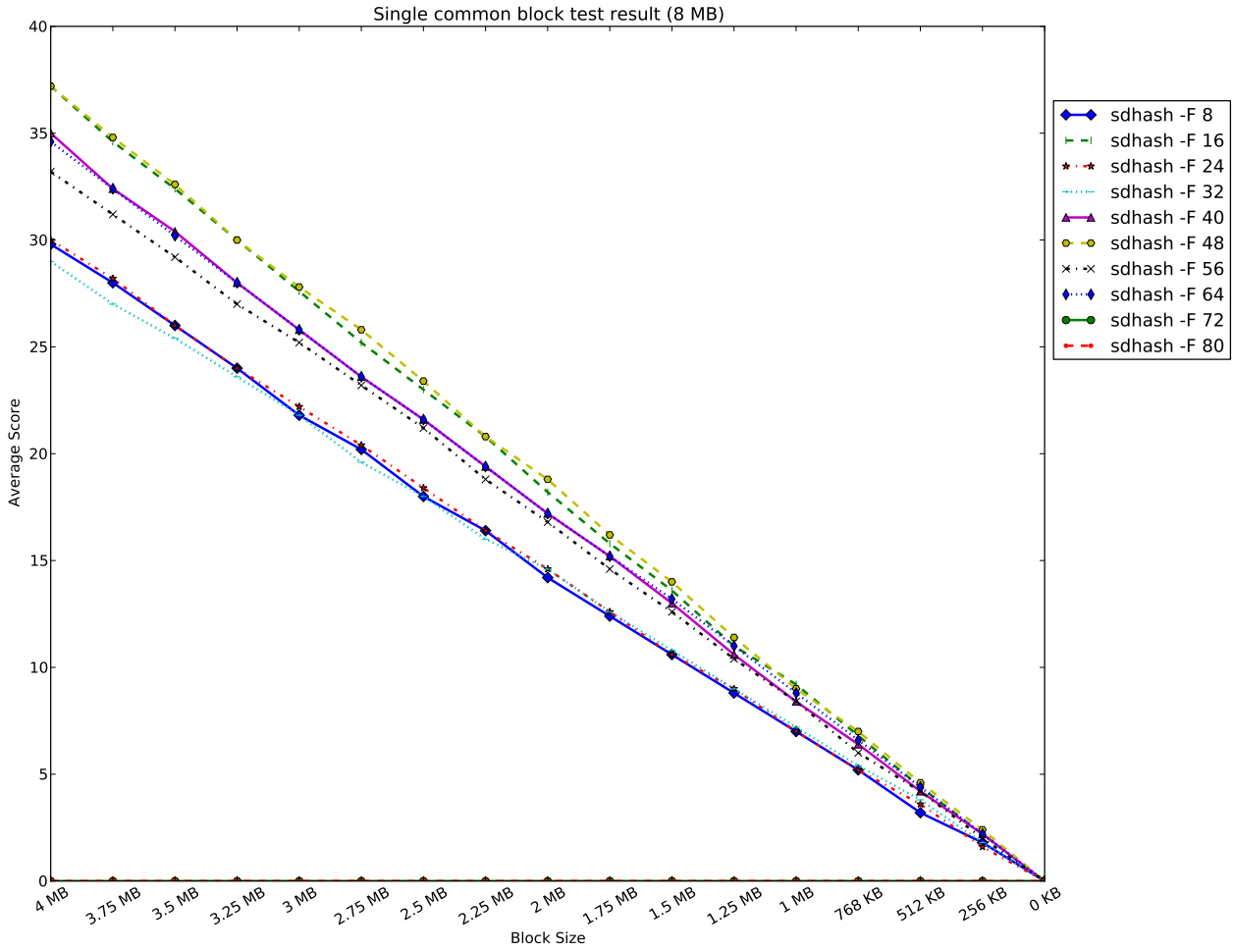


Figure 4.18: Average similarity scores for comparison of randomly generated 8-MiB files with a single common block of decreasing size, measured using *popularity threshold* values between 8 and 80. `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

Random-noise Resistance Test Results

Results for the random-noise resistance tests across all values were tightly grouped, and followed the same general trends as previously described: the scores fall most quickly at the left edge of the graph, then decelerate in their decline before reaching zero. Though all results fall within a narrow range, `sdhash -F 64` stands out as having a slight advantage in both Figures 4.19 and 4.20.

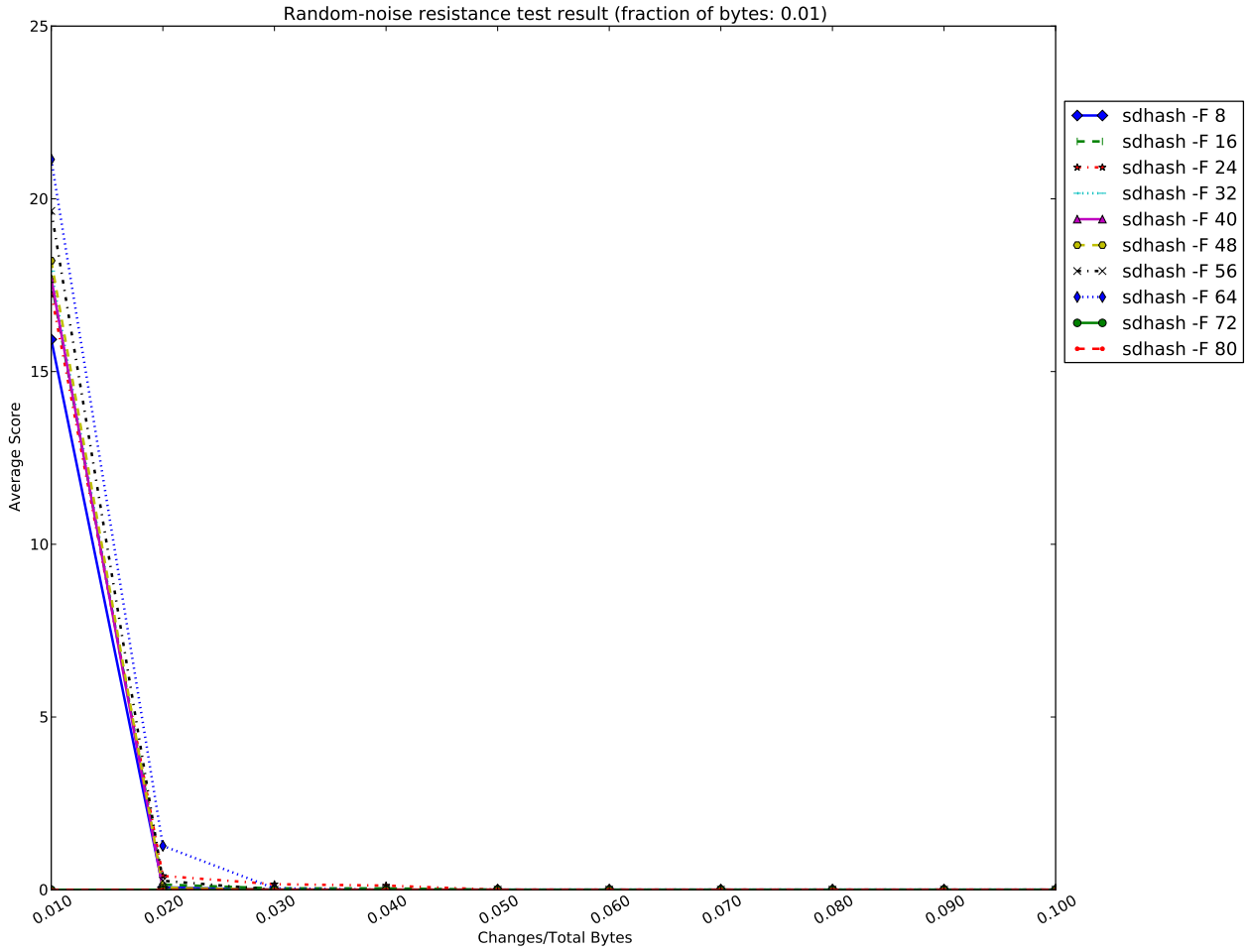


Figure 4.19: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using *popularity threshold* values between 8 and 80 (number of transformations = $\frac{1}{100}$ of total bytes in original file). `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

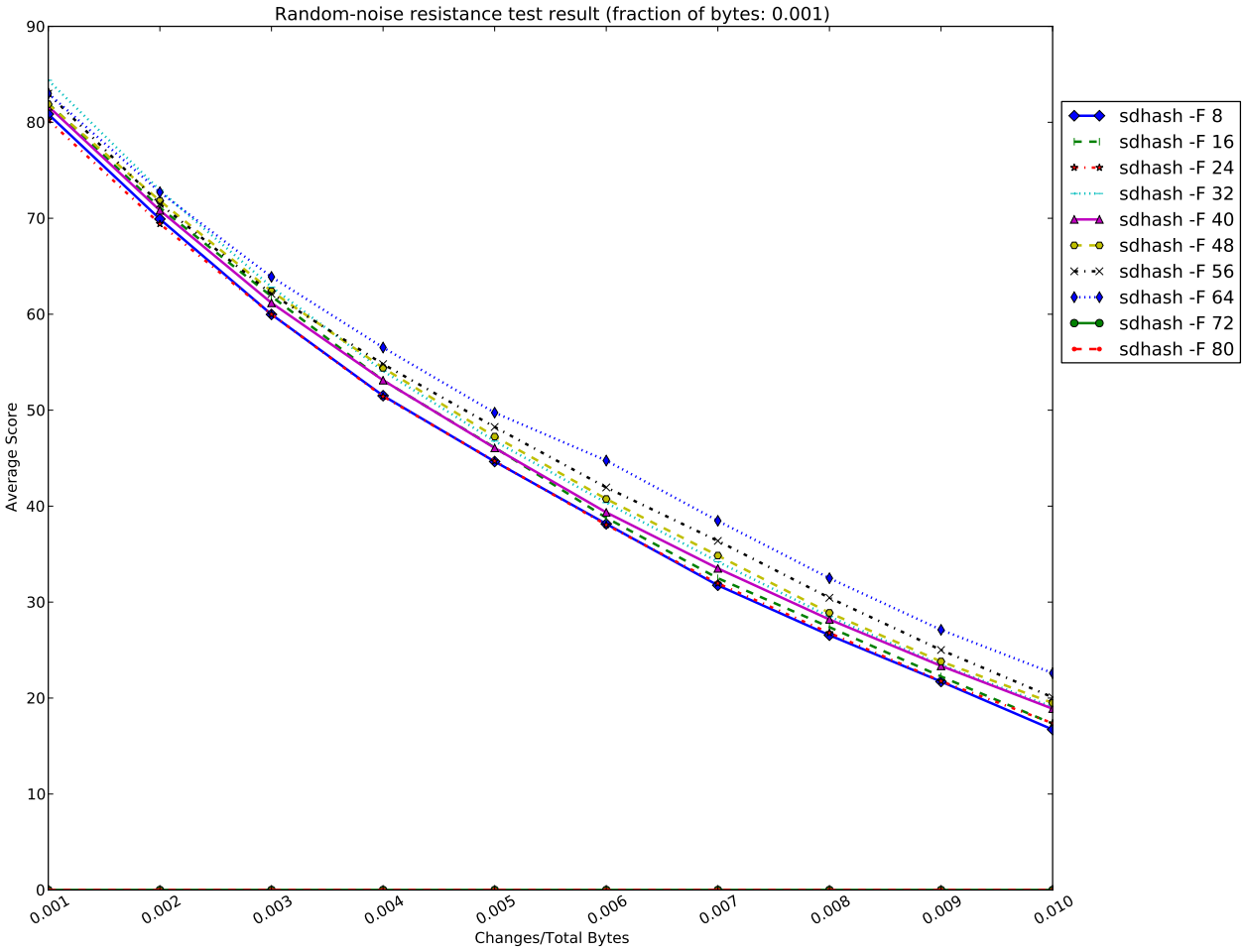


Figure 4.20: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using *popularity threshold* values between 8 and 80 (number of transformations = $\frac{1}{1000}$ of total bytes in original file). `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

Alignment Test Results

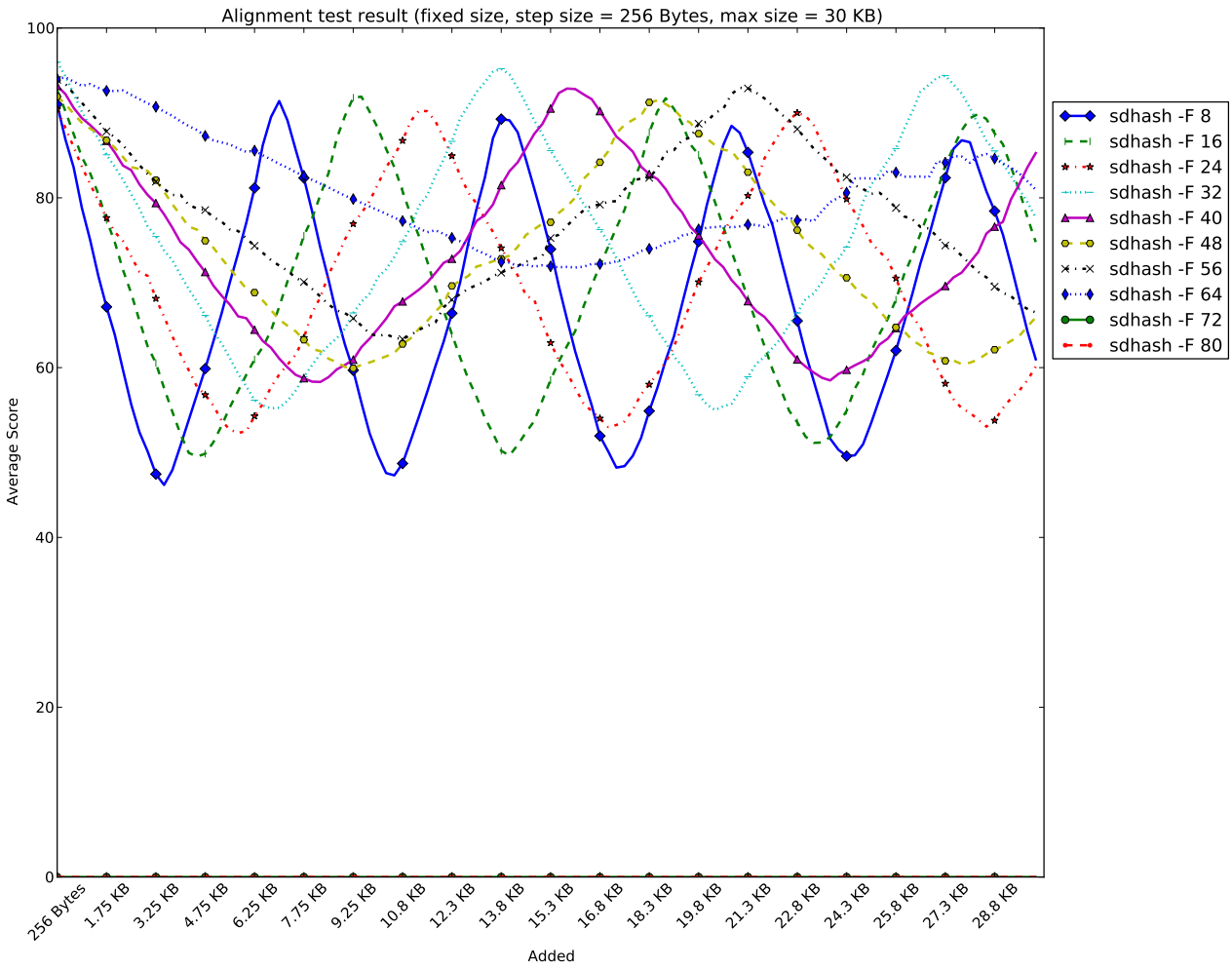


Figure 4.21: Average similarity scores for comparison of files against copies of themselves with chunks of random data prepended, measured using *popularity threshold* values between 8 and 80 (chunk size = 256 bytes). `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

In regard to the *popularity threshold* parameter, the results of the alignment test proved the most illuminating. Looking at Figure 4.21, we see a pattern of oscillations similar to that observed in the results of the *sd score scale* parameter. In contrast to those results, however, variations in the *popularity threshold* do not correspond simply to a linear increase in the amplitude of the curve. Rather, higher threshold values expand the period of the wave while reducing its amplitude. Thus, the setting that most consistently detects containment is `sdhash -F 64`.

A driving factor behind the trend of increasing period is likely quite straightforward: increasing the *popularity threshold* causes fewer features to be chosen. This, in turn, slows the process of shifting the Bloom filters out of alignment, which is the source of the oscillation in the score. The reduction in amplitude is more difficult to explain. We know from the compression tests hat a high *popularity threshold* causes fewer features to be selected, thus filling fewer Bloom filters and creating shorter signature files. One possible explanation, then is that the misalignment in the Bloom filters is counter balanced by a steadily high score in the last filter (which will continue to match at the same rate). A high scoring filter would exert more influence for short files with few total filters. If this is the case, it may be that scores from the smallest files in our data set are driving up the average scores reported in this figure. Regardless, further examination of the actual filters produced during this test is needed to verify this hypothesis.

Fragment Detection Test Results

The most salient feature of the fragment detection tests, shown in Figures 4.22 and 4.23 is the divergence of `sdhash -F 64` from the other parameter settings. In both graphs, and especially Figure 4.22, we see a sharp decrease in average score as the fragment size decreases. This phenomenon appears to corroborate our hypothesis from Section 4.3.2 that the number of Bloom filters created and the SF_{score} value of the final filter may be dominating the average similarity

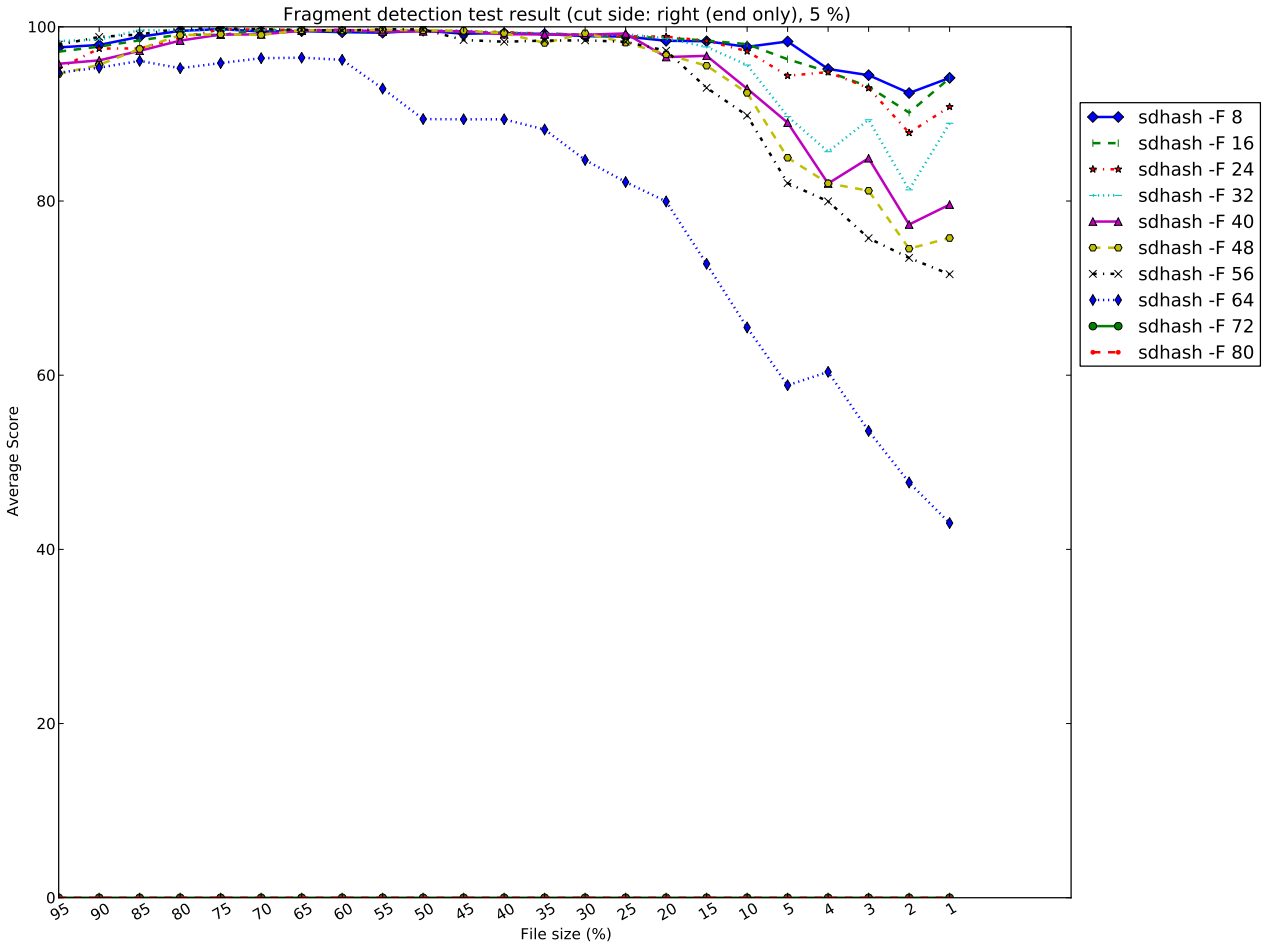


Figure 4.22: Average similarity scores for comparison of files against copies of themselves with slices removed from the tail, measured using *popularity threshold* values between 8 and 80 (slice size = 5% of file size until 5% remains, then 1%). `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

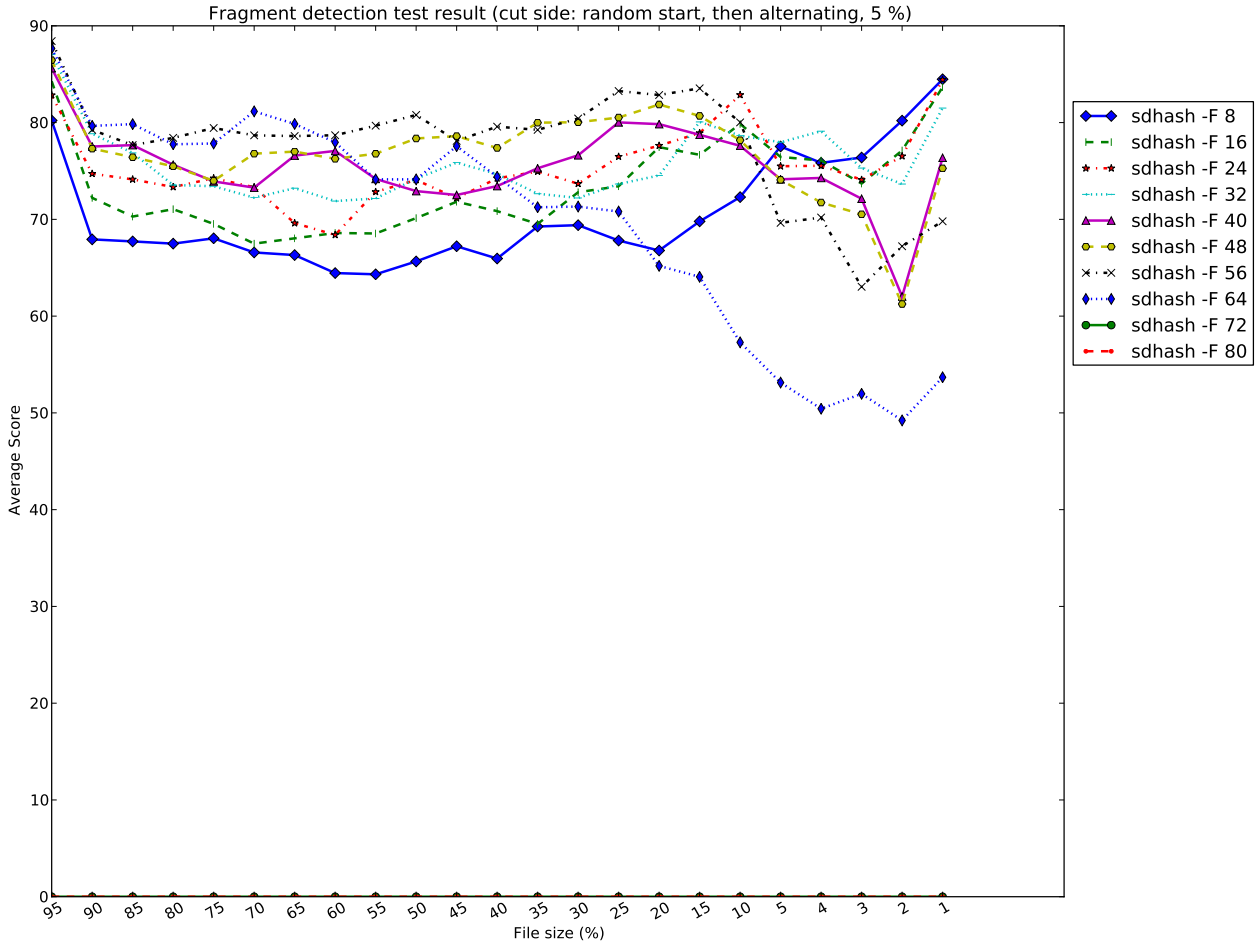


Figure 4.23: Average similarity scores for comparison of files against copies of themselves with slices removed from alternating ends, measured using *popularity threshold* values between 8 and 80 (slice size = 5% of file size until 5% remains, then 1%). `sdhash -F 16` matches Roussev's settings. `sdhash -F 72` and `sdhash -F 80` are flush with the x axis.

score. As the target file is sliced from the right side into smaller and smaller fragments, it will produce smaller signatures containing fewer filters. As in the alignment tests, a high popularity score exacerbates this situation by reducing the signature size even further. The slicing process damages the final Bloom filter, and as this receives greater weight it drives the score down.

This outcome is problematic for `sdhash -F 64`. Although this setting was clearly the best choice for the alignment tests and had a minor advantage in the random-noise resistance tests,

the pattern it shows in this test does not meet the criteria of high, consistent scoring that we expect from a measure of containment. Furthermore, although there is some suggestion of the kind of decline associated with a commonality measure when the common material is reduced, we believe that this is merely a coincidental side-effect of damage to the final filter.

Overall, no value emerged as clearly superior in this experiment. The main effect of changing the *popularity window* appears to be tied to increasing or decreasing the total number of features selected. Because the results are not as pronounced as those seen in the *sd score scale* experiment, and because the ramifications of altering the threshold are not as obvious, we refrain from recommending a setting. Additional experimentation in combination with changes in the size of the *popularity window* may reveal more conclusive outcomes.

4.3.3 Popularity Window Size

Perhaps owing to the interrelationship between the two parameters, our tests with a variety of *popularity window* sizes had much in common with the results from the *popularity threshold* experiment. Intriguingly, we see at least two settings, `sdhash -P 16` and `sdhash -P 144`, that challenge the conclusion that the quantity of features selected primarily determines algorithm behavior.

Since we use the default *popularity threshold* of 16, window sizes of one and fifteen select no features and were included only to show the lower limit of the range. These settings produce only zero scores.

Single Common Block Results

Figures 4.24, 4.25 and 4.26 demonstrate behavior nearly identical to that of the single-common-block tests for the *popularity threshold* experiment. Each separate run of the test at different file sizes results in what appears to be a random shuffling of the results. As in the previous experiment, we expect that this may be a result of the underlying random data. More work tracking the slopes of the lines in different runs is needed to determine if any patterns occur.

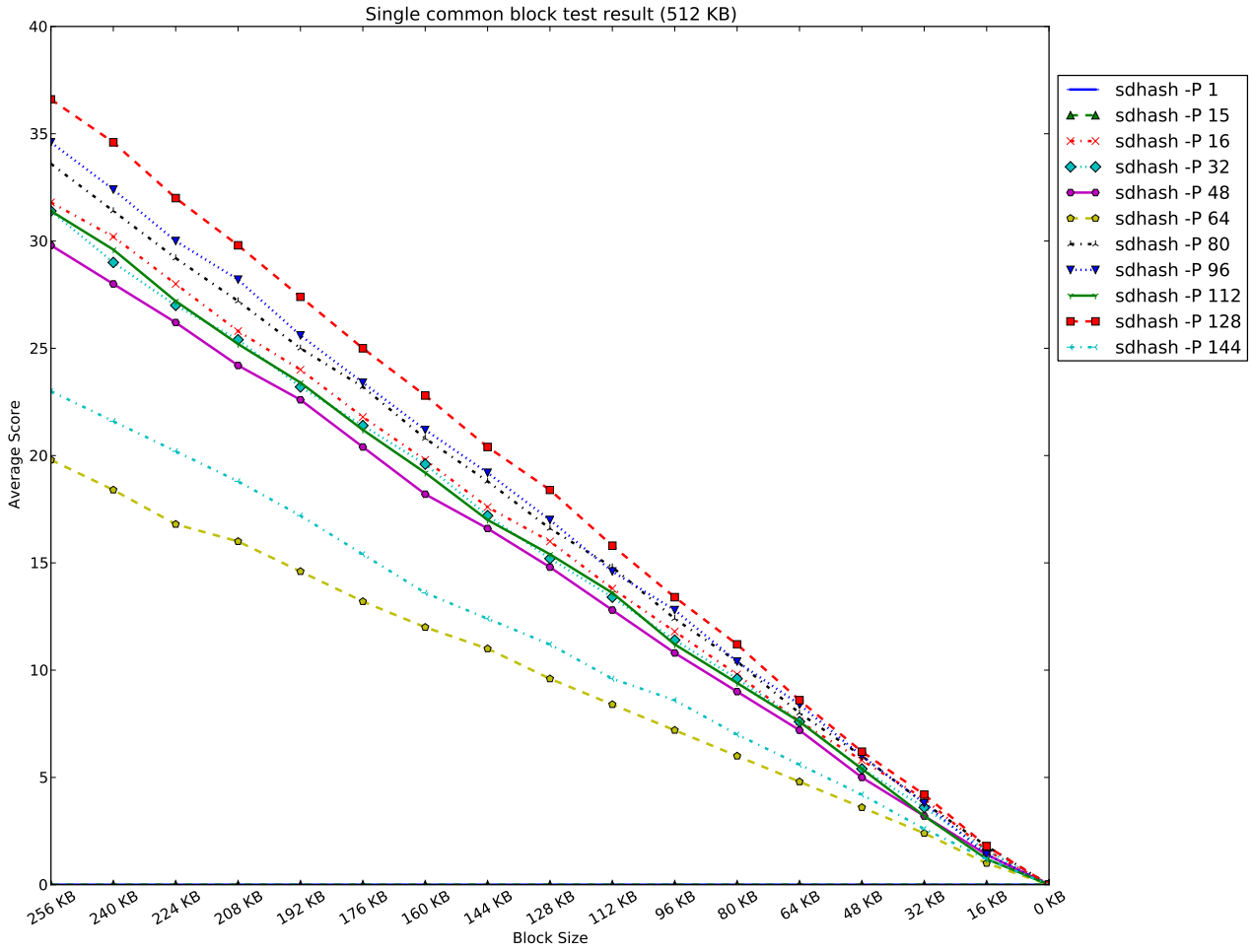


Figure 4.24: Average similarity scores for comparison of randomly generated 512-KiB files with a single common block of decreasing size, measured using *popularity window* sizes between 1 and 160. `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

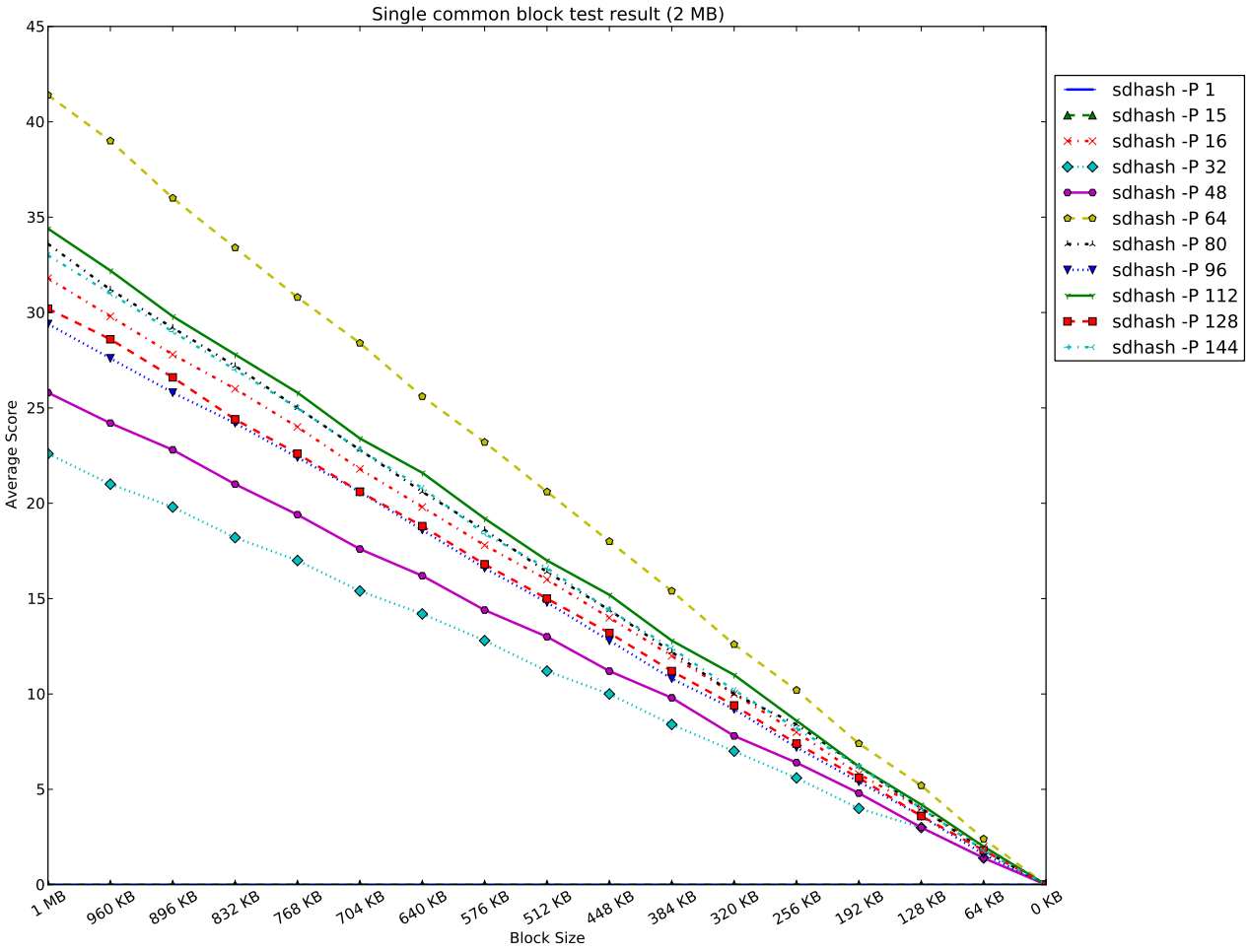


Figure 4.25: Average similarity scores for comparison of randomly generated 2-MiB files with a single common block of decreasing size, measured using *popularity window* sizes between 1 and 160. `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

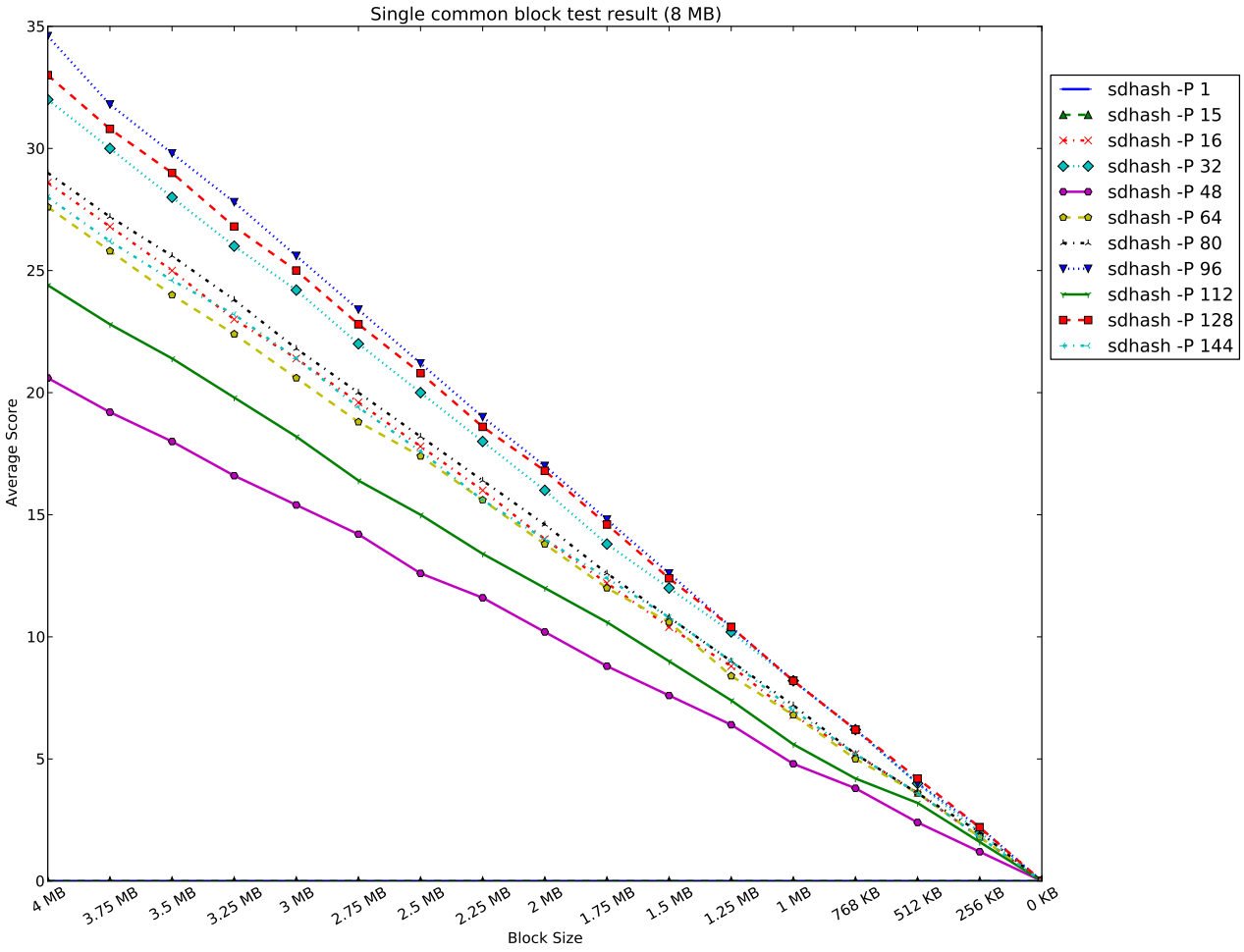


Figure 4.26: Average similarity scores for comparison of randomly generated 8-MiB files with a single common block of decreasing size, measured using *popularity window* sizes between 1 and 160. `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

Random-noise Resistance Test Results

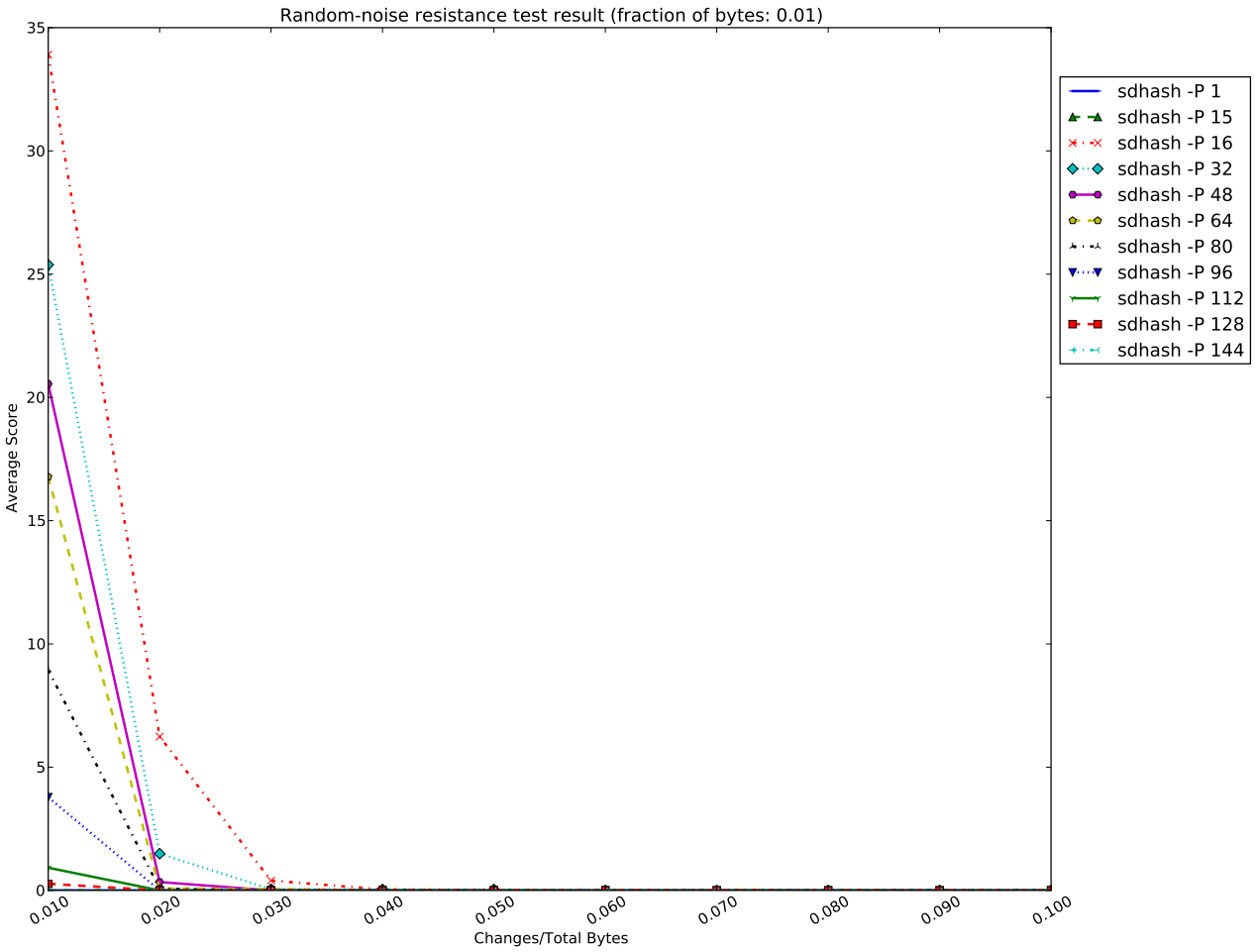


Figure 4.27: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using *popularity window* sizes between 1 and 160 (number of transformations = $\frac{1}{100}$ of total bytes in original file). `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

At first glance the random-noise resistance test appears to directly contradict conclusions derived from the *popularity threshold* experiments previously. In that experiment, we observed that settings corresponding to the smallest signature sizes had some advantage in resisting damage caused by random edits. Although this holds true for `sdhash -P 16`, which gives by far the best results (see Figure 4.27), the ranking of other settings follows their parameter value

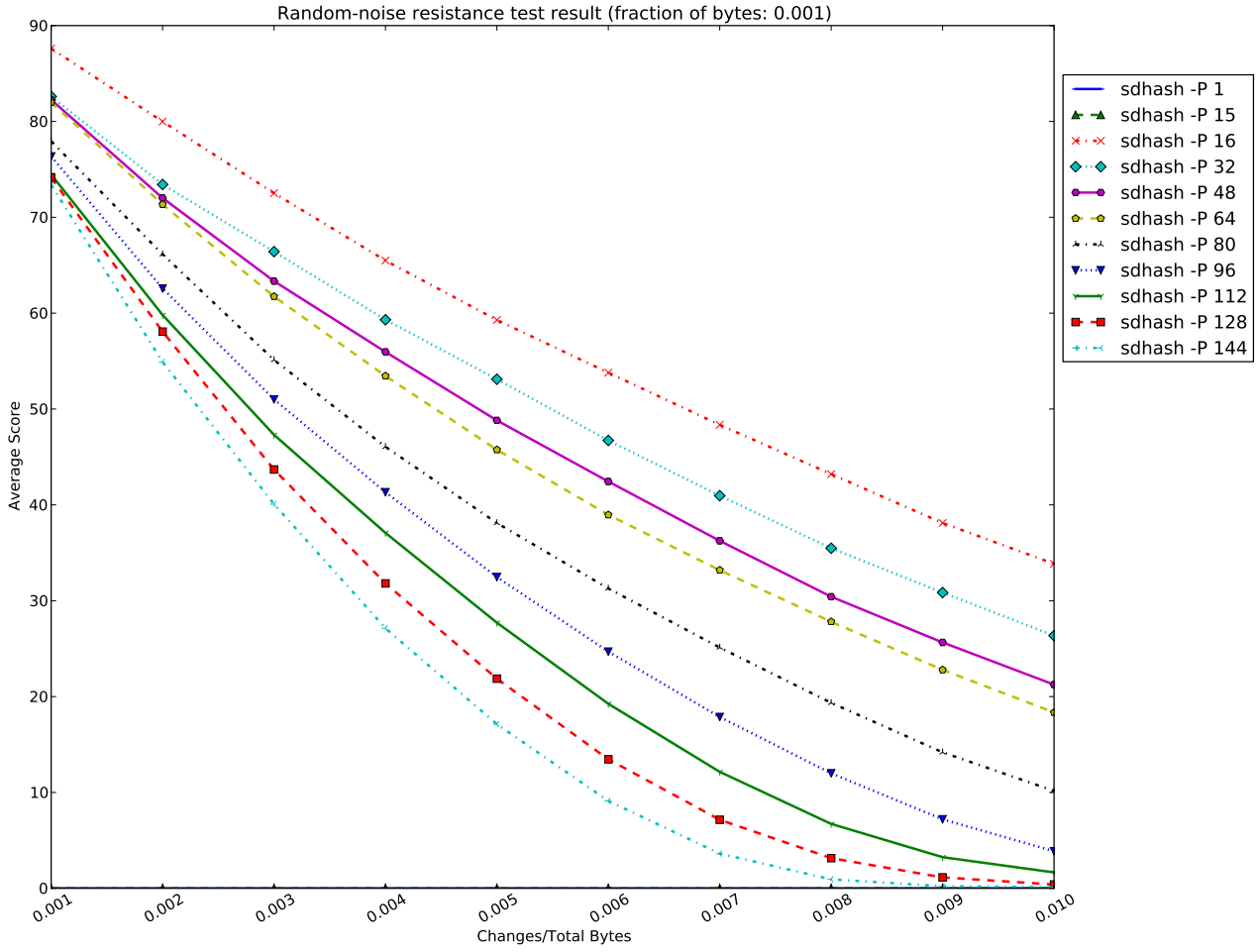


Figure 4.28: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using *popularity window* sizes between 1 and 160 (number of transformations = $\frac{1}{1000}$ of total bytes in original file). `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

rather than the signature size to which it corresponds.

As Figure 4.28 shows clearly, the relations between parameter settings and results is as linear as that seen in the *sd score scale* experiment, despite a non-linear relationship between these settings and signature size. We hypothesize that shrinking the size of the *popularity window* limits the range of the damage that can be done by a random edit. In settings that produce a high

feature selection rate, such as `sdhash -P 32`, the increased number of features compensates for the increased probability of losing a feature during a given edit. In the special case of `sdhash -P 16`, however, even though fewer features are selected as a result of the fact that a feature can only be selected if it receives a perfect score (i.e., groupings are not possible), the feature depends on only a very small surrounding area. The likelihood of a random edit striking this area remains low. Put another way, forcing the window size to correspond to the *popularity threshold* ensures only one feature occurs in the span covered by that window as it slides across the feature. When this is not the case, even though more features are selected, a random edit could eliminate several of them at once.

Alignment Test Results

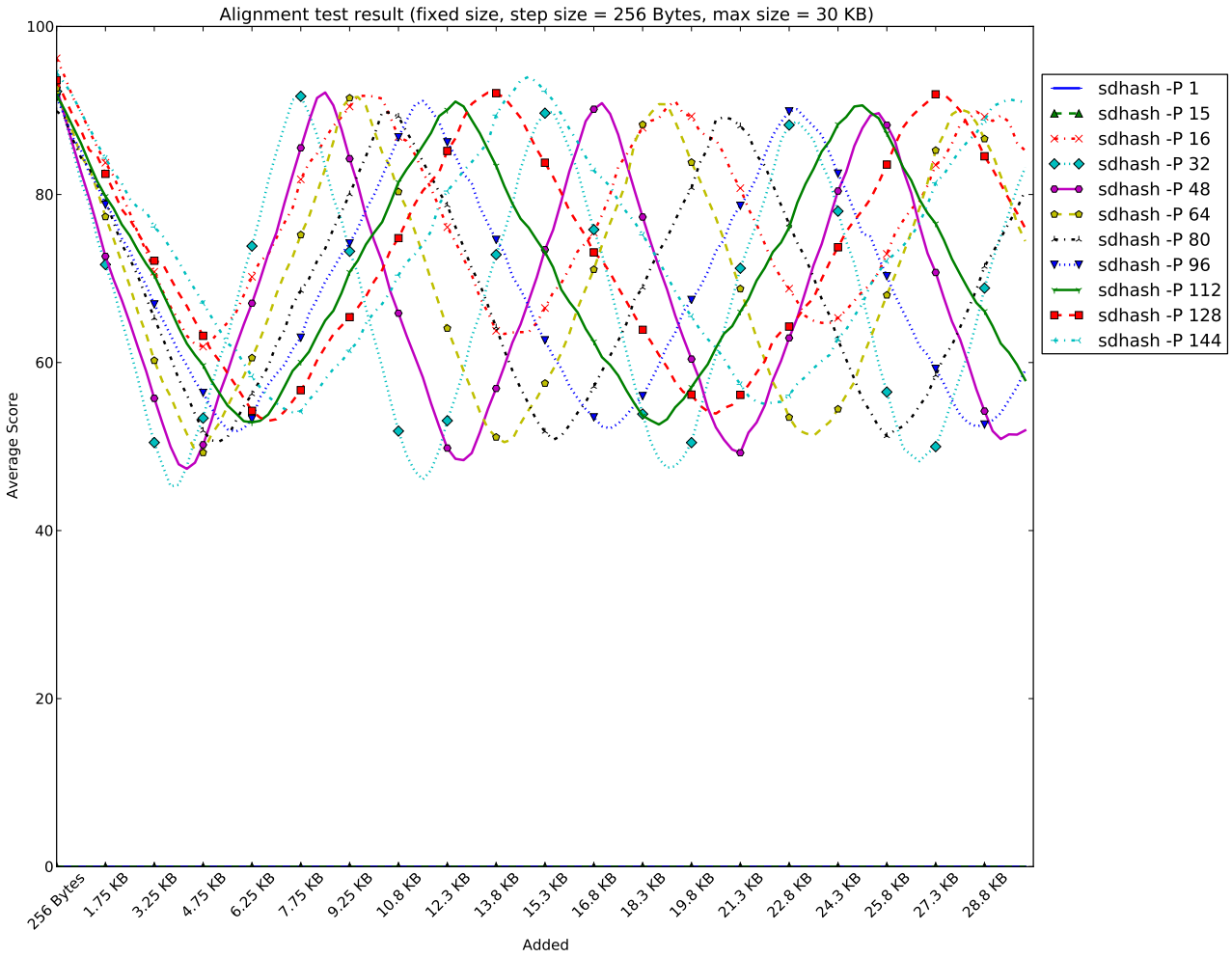


Figure 4.29: Average similarity scores for comparison of files against copies of themselves with chunks of random data prepended, measured using *popularity window* sizes between 1 and 160 (chunk size = 256 bytes). `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

At first glance, the results of the alignment test (Figure 4.29) appear to continue the trend introduced in the *popularity threshold* experiment, in which settings that cause a higher rate of feature selection (and therefore longer signatures) correspond to shorter periods and larger amplitudes. `sdhash -P 16` is a clear outlier in this regard, showing a mid-range period in combination with the smallest amplitude.

We resort to our previous hypothesis regarding amplitude: the fewer features selected, the more

the score of the final Bloom filter is able to buoy the others, especially for small files. In this case, though, it is unclear why the period of the curve is between the period of `sdhash -P 64` and `sdhash - P 80`. This seems to imply that the parameterization is choosing features just as quickly as settings that result in much longer signatures. The only theory we can offer in this regard is that it may be that `sdhash -P 16` samples at a different rate for regular and random data. The period is only reflective of the latter, so this may account for some discrepancy. A more detailed look at the Bloom filters created may shed additional light on this outcome. Whatever the underlying reason, `sdhash -P 16` appears to be the best choice so far for resistance to both random noise and alignment shifts.

Fragment Detection Test Results

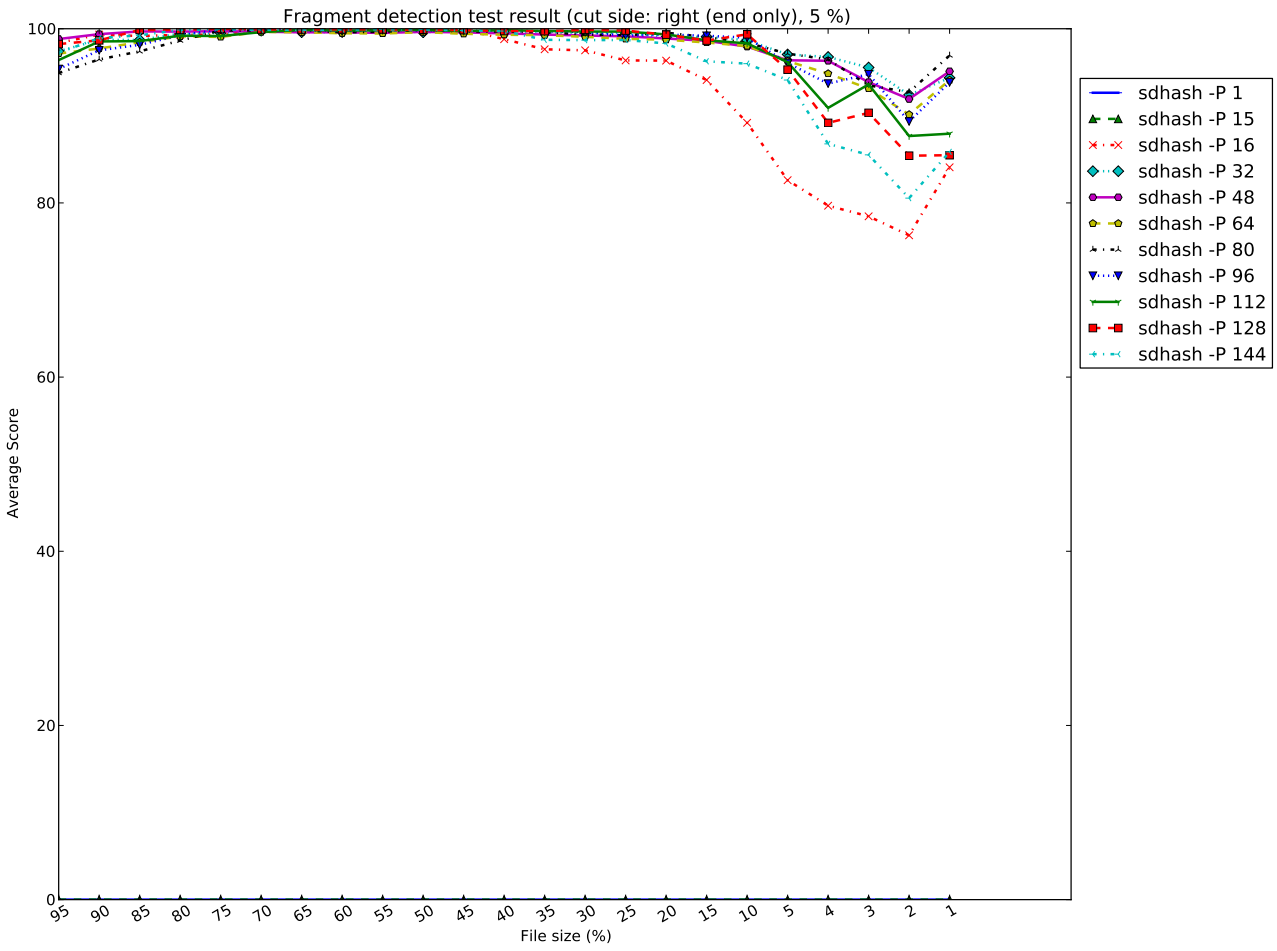


Figure 4.30: Average similarity scores for comparison of files against copies of themselves with slices removed from the tail, measured using *popularity window* sizes between 1 and 160 (slice size = 5% of file size until 5% remains, then 1%). `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

The curves produced by each parameter setting in the fragment detection test are closely grouped together. As in the *popularity threshold* experiment, settings producing low feature-selection rates are at a disadvantage. `sdhash -P 64` and `sdhash -P 144` dip below the other curves shown in Figure 4.30. Although `sdhash -P 16` begins high in Figure 4.31, it drops below its neighbors as the fragment size decreases. As mentioned, we believe this effect to be the result of a damaged final filter and reduces number of filters to average against it. Even the settings

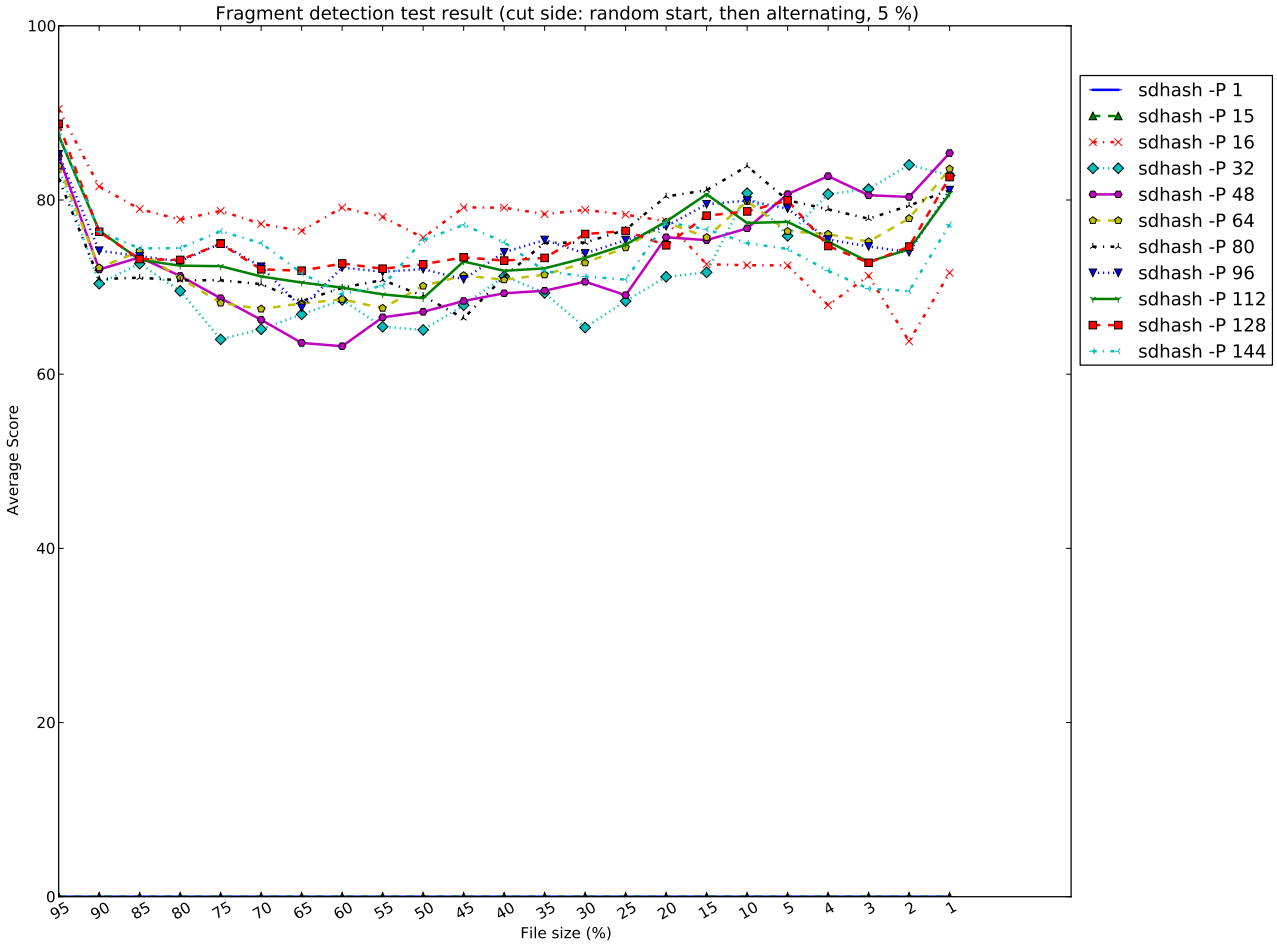


Figure 4.31: Average similarity scores for comparison of files against copies of themselves with slices removed from alternating ends, measured using *popularity window* sizes between 1 and 160 (slice size = 5% of file size until 5% remains, then 1%). `sdhash -P 64` matches Roussev's settings. `sdhash -P 1` and `sdhash -P 15` are flush with the x axis.

corresponding to the lowest feature-selection rates did not produce signatures as short as those from `sdhash -F 64`, and generally the output was considerably more stable.

Thus, although the performance for `sdhash -P 16` is not ideal for fragment detection, neither is it unacceptably poor. In light of the fact that this setting had a clear advantage in the random-noise resistance and alignment tests, we argue that it is the best choice among the options explored. Because of the interaction with *popularity threshold*, and in particular several

correlations suggested by parameter settings where two were set equal to each other (at values of 16 and 64), we posit that further optimization might be achieved by a combinatoric exploration of the two settings together.

4.3.4 Entropy Rank

The large number of possible values in the *entropy rank* table prohibited exhaustive testing. As a result, we were limited to a set of discrete tables, each designed according to an arbitrarily chosen method. Our hope was to either to validate the table created by Roussev, or to discover patterns that might better direct future investigation.

While we could not cover enough of the parameter space to make a convincing argument for a parameter setting at this time, the results of these experiments were among the most illuminating—with respect to both the role of the parameter in the algorithm and to the relationship between the algorithm and the properties of its input data. For convenience, we rely on the reference codes listed in Table 3.3 throughout our discussion.

A few general trends can be observed. ENT_0 and ENT_5 are nearly identical in all tests except fragment detection. Likewise, ENT_6, ENT_7 and ENT_8 follow each other closely for all tests, and differ from the other tables significantly when the tests involve random data. Finally, it is worth pausing to underscore an aspect of the results that is so clearly demonstrated as to be in danger of being overlooked: none of the various arbitrarily chosen tables prevented the algorithm from functioning altogether. Even in cases were certain aspects of its behavior were altered, there is still a remarkable resemblance between all settings and ENT_0 (Roussev's table) in almost all cases.

Single Common Block Results

Figures 4.32, 4.33 and 4.34 exhibit a number of major characteristics warranting discussion: First, as noted, ENT_0 and ENT_5 are exactly in line with each other. This means that ENT_5 follows the "factory default" behavior of *sdhash* for this test. Since the main difference between the two tables is the introduction of additional null scores in the lower section of ENT_5, we can infer that the features used in this test had higher H_{norm} values. Given the fact that the tests use randomly generated files this result is as predicted.

Second, curves produced by parameter setting ENT_0 to ENT_5 are roughly consistent with the expectations of a commonality test. All produce a smooth sloped line that reaches zero when the shared common material disappears. However, ENT_1 through ENT_44 change their arrangement from one test to the next. The factory default settings of *sdhash* have the most consistent curve across all three tests. Further experimentation is needed to verify that this is true for multiple runs against the file size and block size.

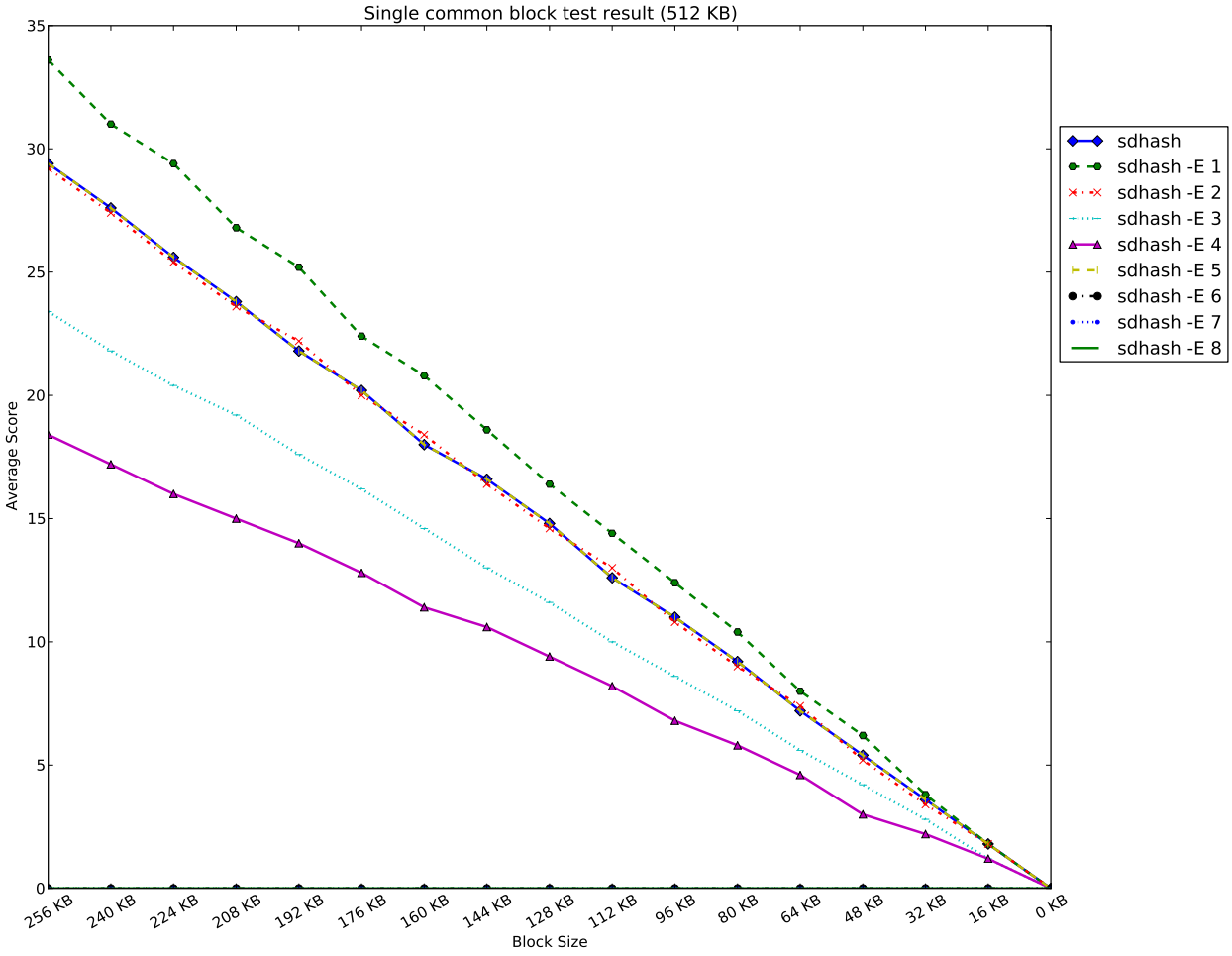


Figure 4.32: Average similarity scores for comparison of randomly generated 512-KiB files with a single common block of decreasing size, measured using 9 distinct *entropy rank* tables. `sdhash` uses Roussev's settings. `sdhash -E 6`, `sdhash -E 7` and `sdhash -E 8` are flush with the x axis.

Finally, ENT_6 through ENT_8 produce scores of zero across all three single-common-block tests. Initially, this may appear to indicate that the algorithm is entirely broken for these settings. As we will see, results of later tests show that this is not the case. Rather, we contend that these tables, which have had the upper range of H_{prec} values replaced by nulls, are unable to select features from randomly generated data, since such data is like to have a high average H_{norm} score. This alteration renders them blind to all randomly generated data. Thus, they are unable to detect a randomly generated common block in two randomly generated files.

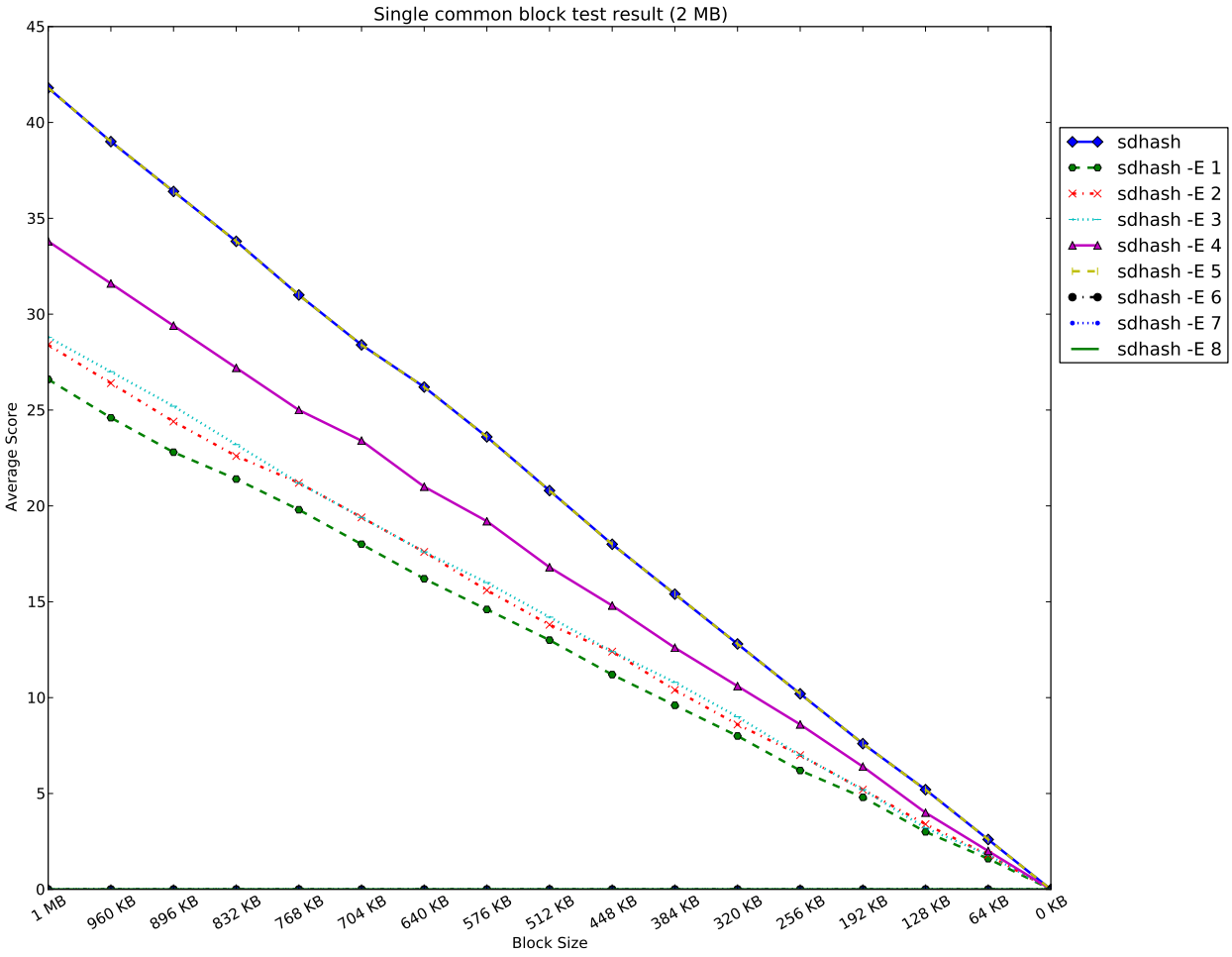


Figure 4.33: Average similarity scores for comparison of randomly generated 2-MiB files with a single common block of decreasing size, measured using 9 distinct *entropy rank* tables. `sdhash` uses Roussev's settings. `sdhash -E 6`, `sdhash -E 7` and `sdhash -E 8` are flush with the x axis.

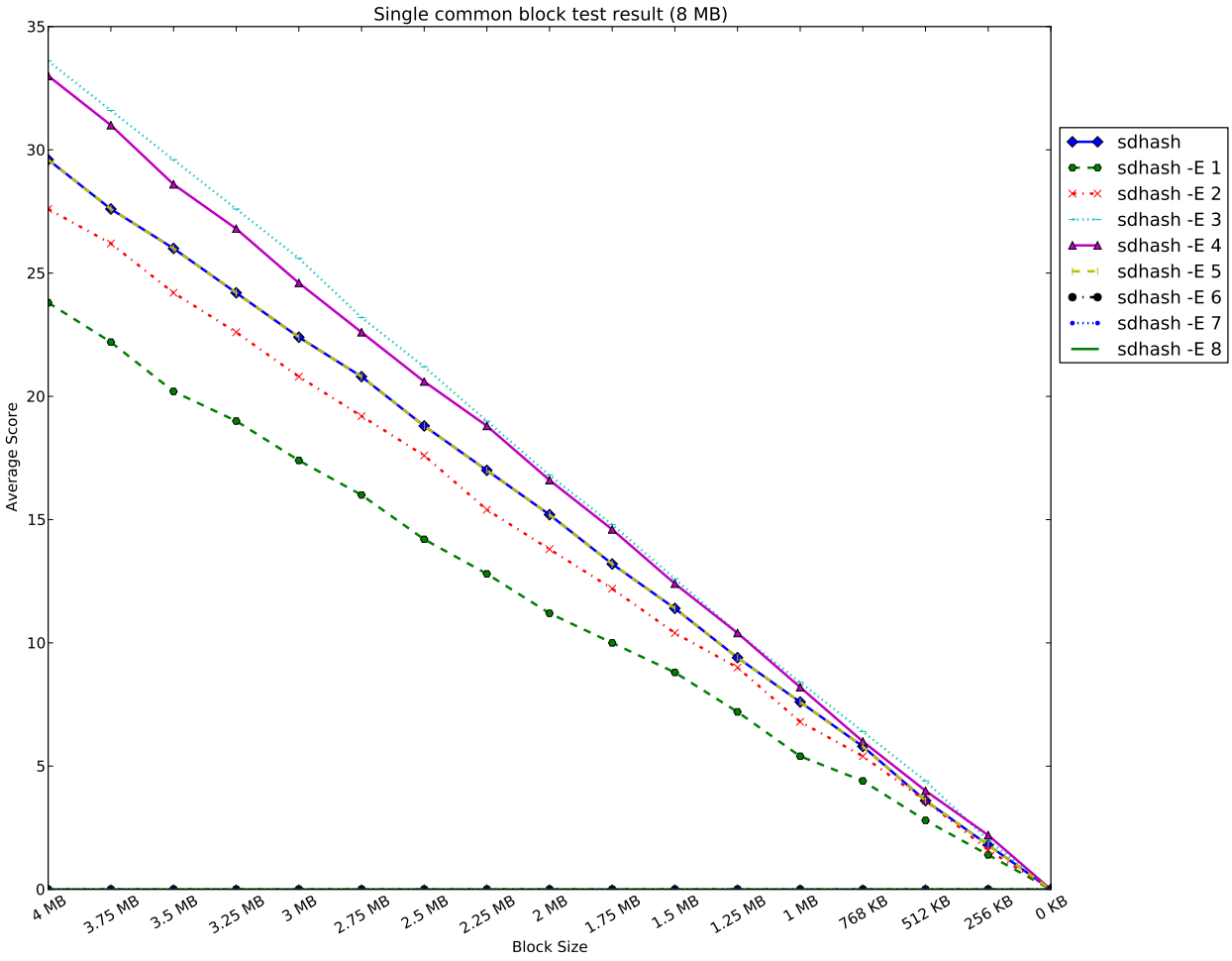


Figure 4.34: Average similarity scores for comparison of randomly generated 8-MiB files with a single common block of decreasing size, measured using 9 distinct *entropy rank* tables. `sdhash` uses Roussev's settings. `sdhash -E 6`, `sdhash -E 7` and `sdhash -E 8` are flush with the x axis.

Random-noise Resistance Test Results

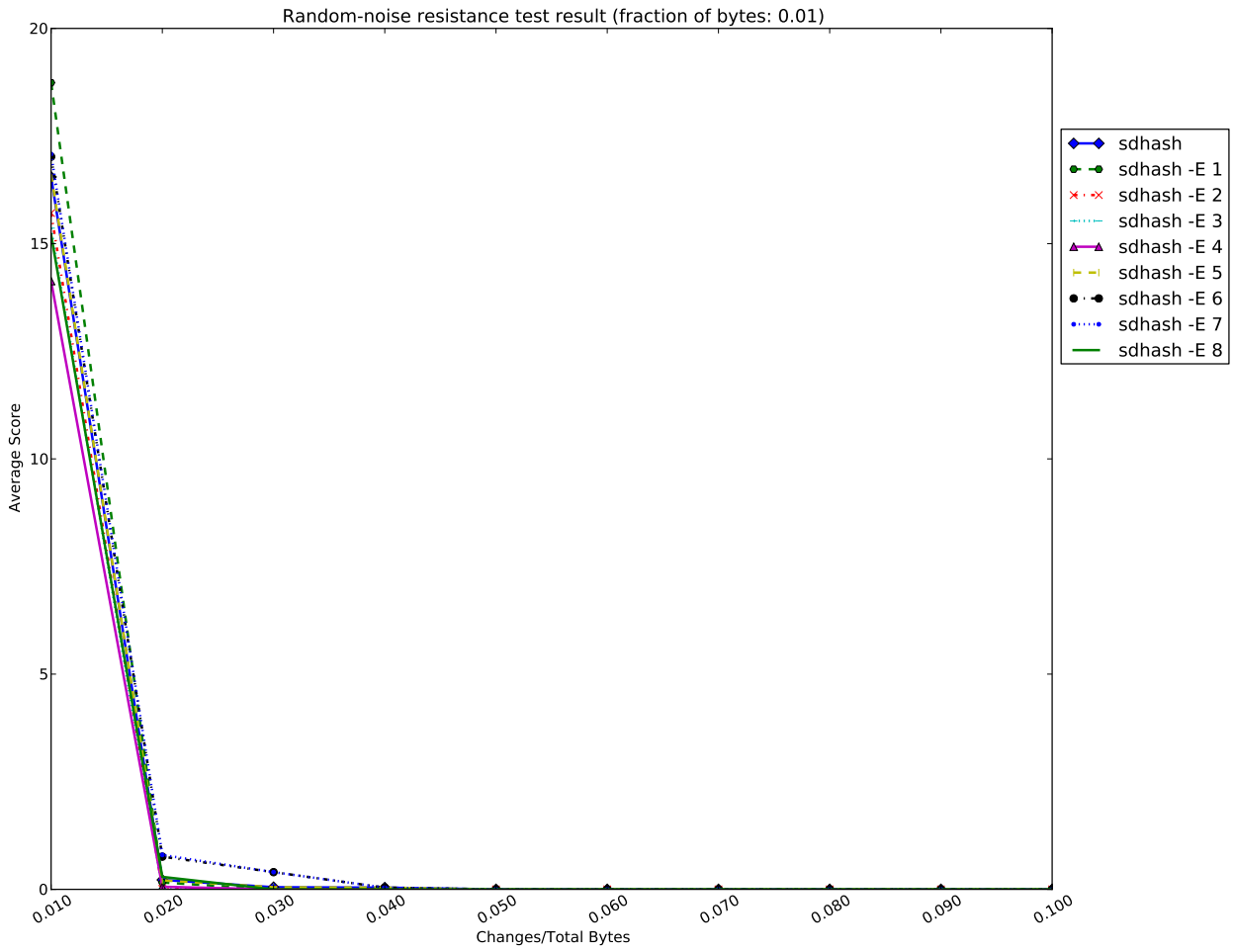


Figure 4.35: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using 9 distinct *entropy rank* tables (number of transformations = $\frac{1}{100}$ of total bytes in original file). `sdhash` uses Roussev's settings.

What is striking about the outcome of the random-noise resistance tests, shown in Figures 4.35 and 4.35 is that all parameter settings appear to be functioning as expected, and closely match the behavior produced by Roussev's settings. Since the random-noise test causes scores to decay by wiping out features, these results strongly suggest that all parameters are still picking features (thus demonstrating that ENT_6 through ENT_8 are not, in fact, completely broken in all scenarios). Because the test compares an object to itself and transforms it uniformly at

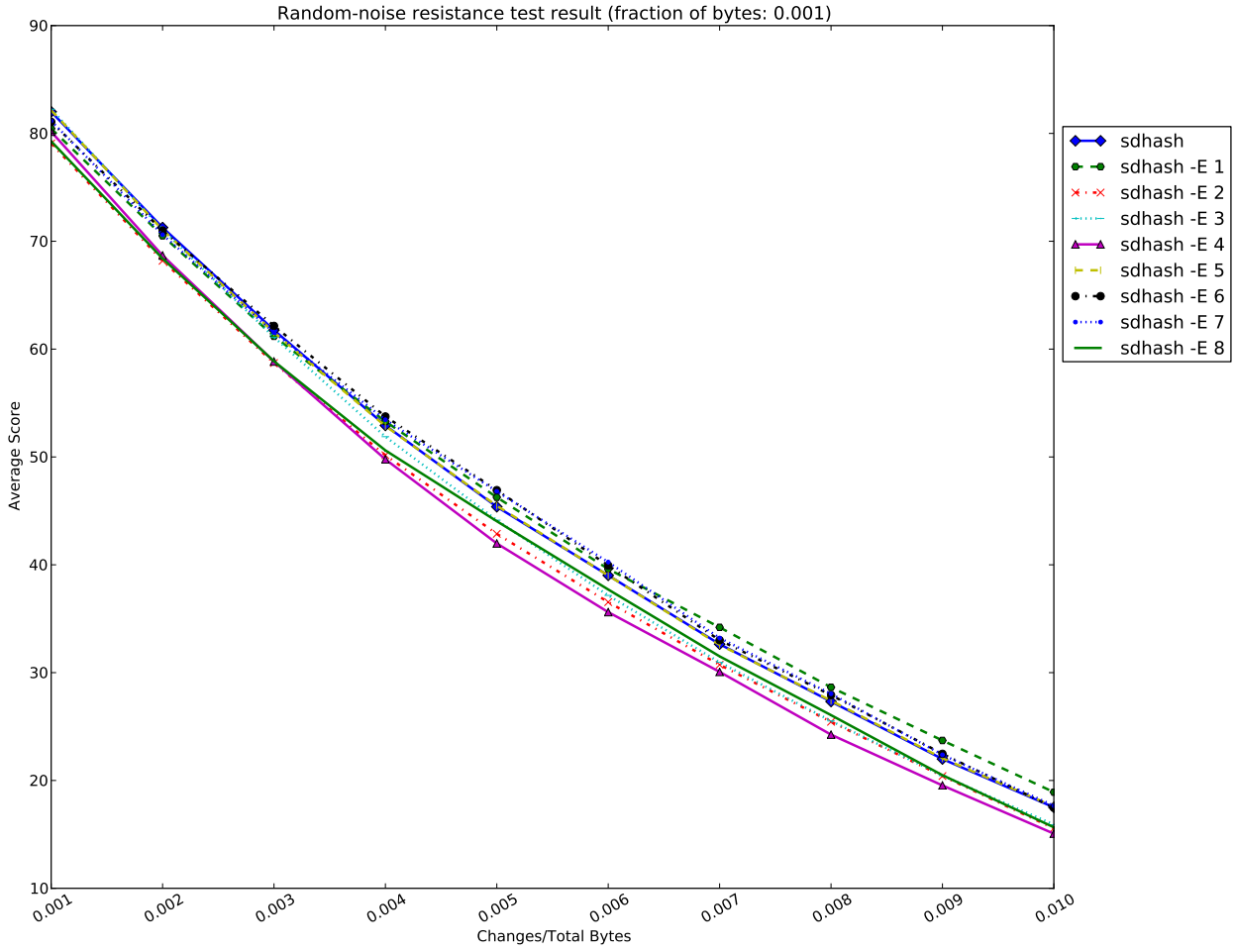


Figure 4.36: Average similarity scores for comparison of files against copies of themselves that are transformed with random byte insertions, deletions and substitutions, measured using 9 distinct *entropy rank* tables (number of transformations = $\frac{1}{1000}$ of total bytes in original file). `sdhash` uses Roussev's settings.

random, each parameterization will compare whatever features it has selected to those that it selects from the transformed file. Thus, the parameterizations need not choose the same (or even related) features to show the same results.

Alignment Test Results

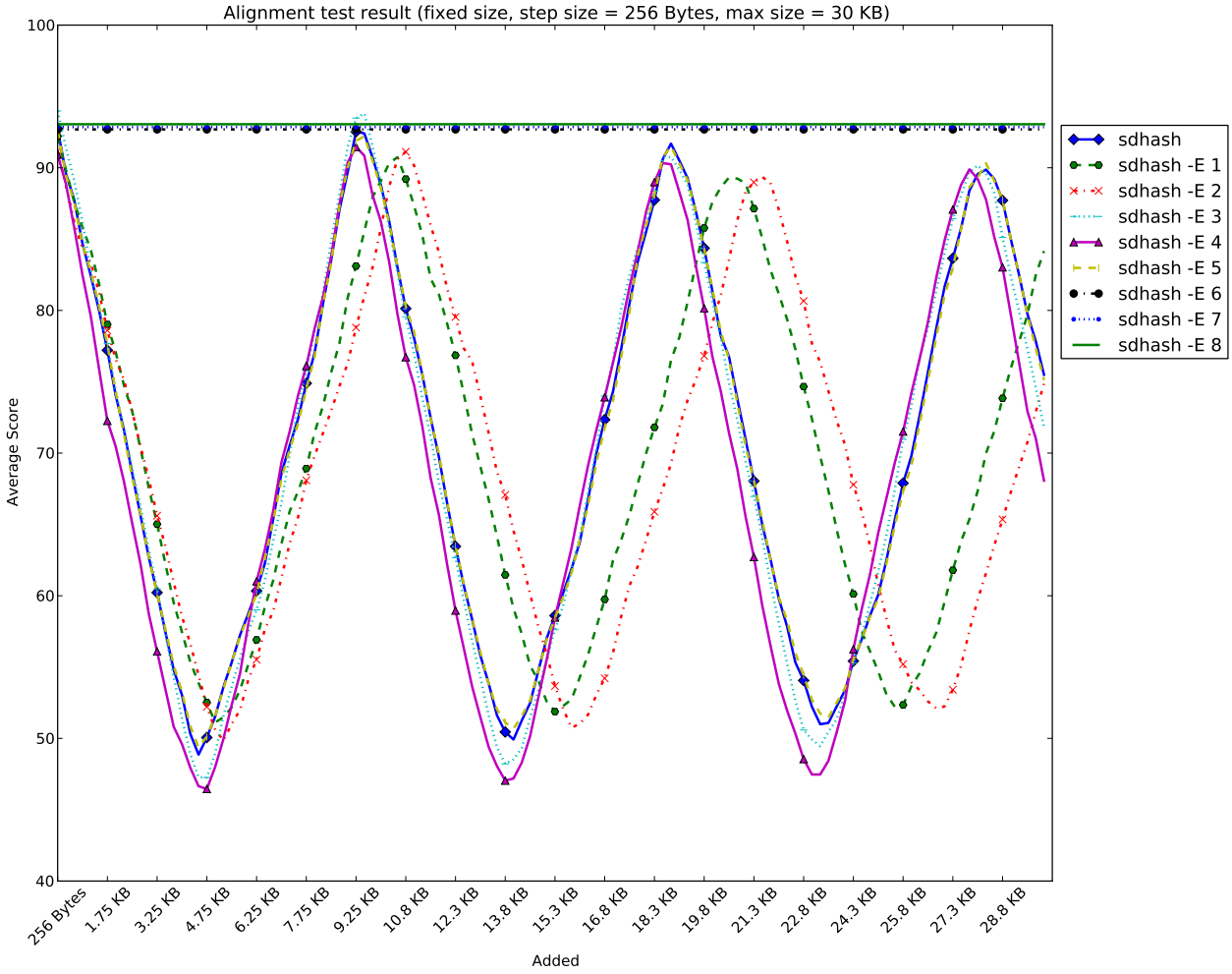


Figure 4.37: Average similarity scores for comparison of files against copies of themselves with chunks of random data prepended, measured using 9 distinct *entropy rank* tables (chunk size = 256 bytes). `sdhash` uses Roussev's settings.

In Figure 4.37, the horizontal lines crossing the top of alignment test represent the same three parameter settings that gave all zeros in the single-common-block test. Here again, it is tempting to assume this flat line indicates that the settings have caused the algorithm to become unresponsive, especially after we have become accustomed to seeing the smooth oscillations that are reproduced by the other parameterizations. With respect to the criteria we have established for measuring containment, however, these three lines are an indication of near-perfect behavior. The relevant question is whether this behavior is an accident or whether *sdhash* is

functioning exactly as it should.

We argue that the latter situation is more likely to be the case. These parameter settings have already demonstrated an insensitivity to random material, as a result of having null values in the table indices for H_{norm} scores in that range. This test works by prepending segments of randomly generated data to the front of one of the compared objects. Usually, this causes features to be generated from that data and added to Bloom filter, producing an alignment shift corresponding to the pattern of oscillating scores. In the case of these *entropy rank* tables, we posit that random data is always mapping to null values, causing no features to be selected from this data. As a result, no alignment shift occurs.

Running these same parameters against an alignment test that uses non-random data (such as sections of other files) would help to confirm this theory. In Section 5.2.3 we suggest adding this option.

Finally, in regard to the rest of the parameter settings shown, we note that the waves are divided loosely into two groupings: ENT_1 and ENT_2, and ENT_0, ENT_3, ENT_4 and ENT_5. This indicates that the tables in these groups are selecting features from random data at approximately the same rate. We note that the main similarity between members of the same group is the "smoothness" of the progression in their respective tables. As the results from the compression tests have already hinted, entropy tables that use a simple increment or decrement choose features at a marginally slower rate than those in which the difference between adjacent H_{prec} values can be greater than one.

Fragment Detection Test Results

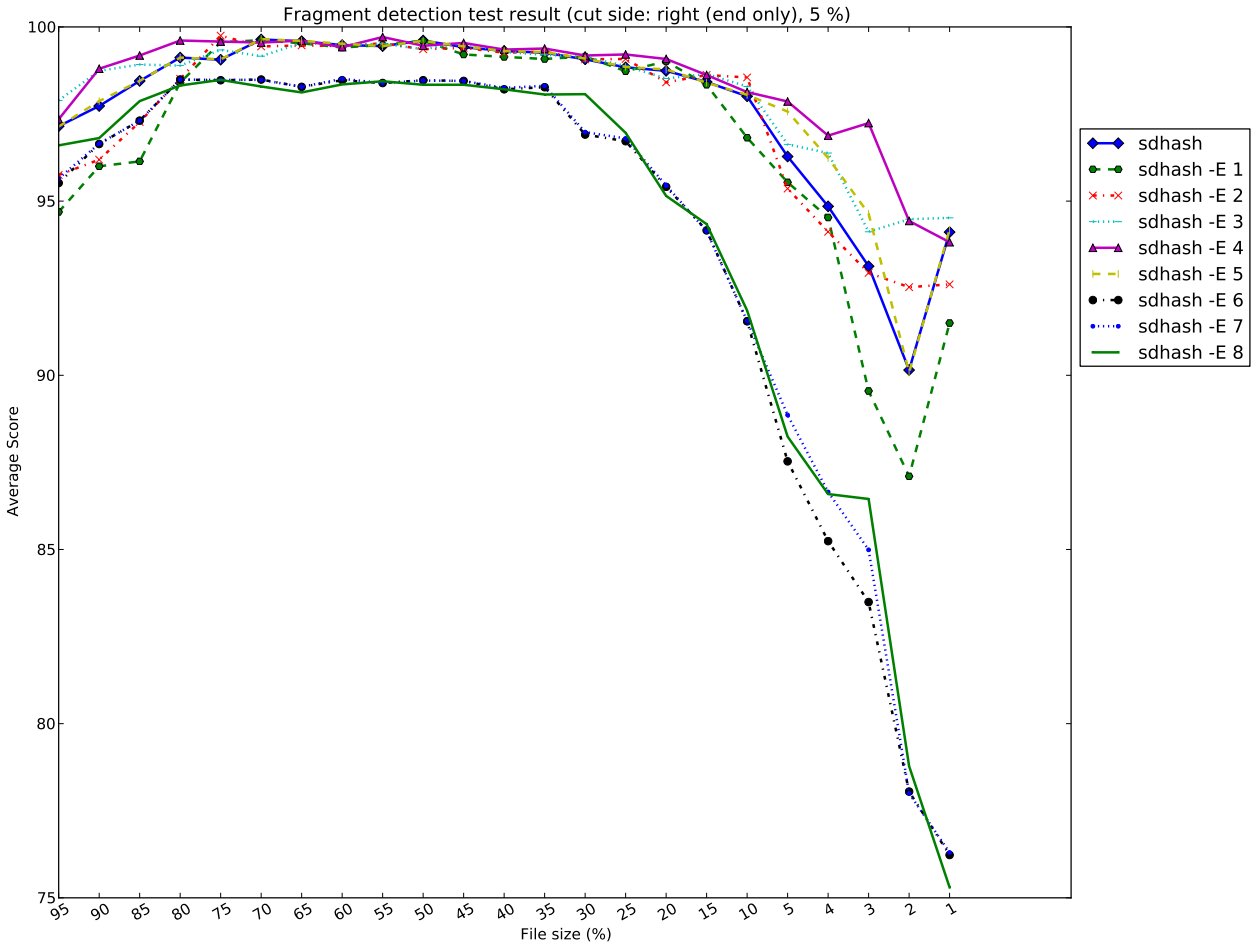


Figure 4.38: Average similarity scores for comparison of files against copies of themselves with slices removed from the tail, measured using 9 distinct *entropy rank* tables (slice size = 5% of file size until 5% remains, then 1%). `sdhash` uses Roussev's settings.

The outcomes of the fragment detection tests, illustrated in Figures 4.38 and 4.39, present a now-familiar division between parameters that maintain a high average score and those that dive as the fragment size diminishes. As before, the parameter settings that produce a dive are all associated with very low feature-selection rates, which cause the damaged ends of the fragments to pull the score toward zero.

Among the other parameters, all demonstrate about the same level of performance with little

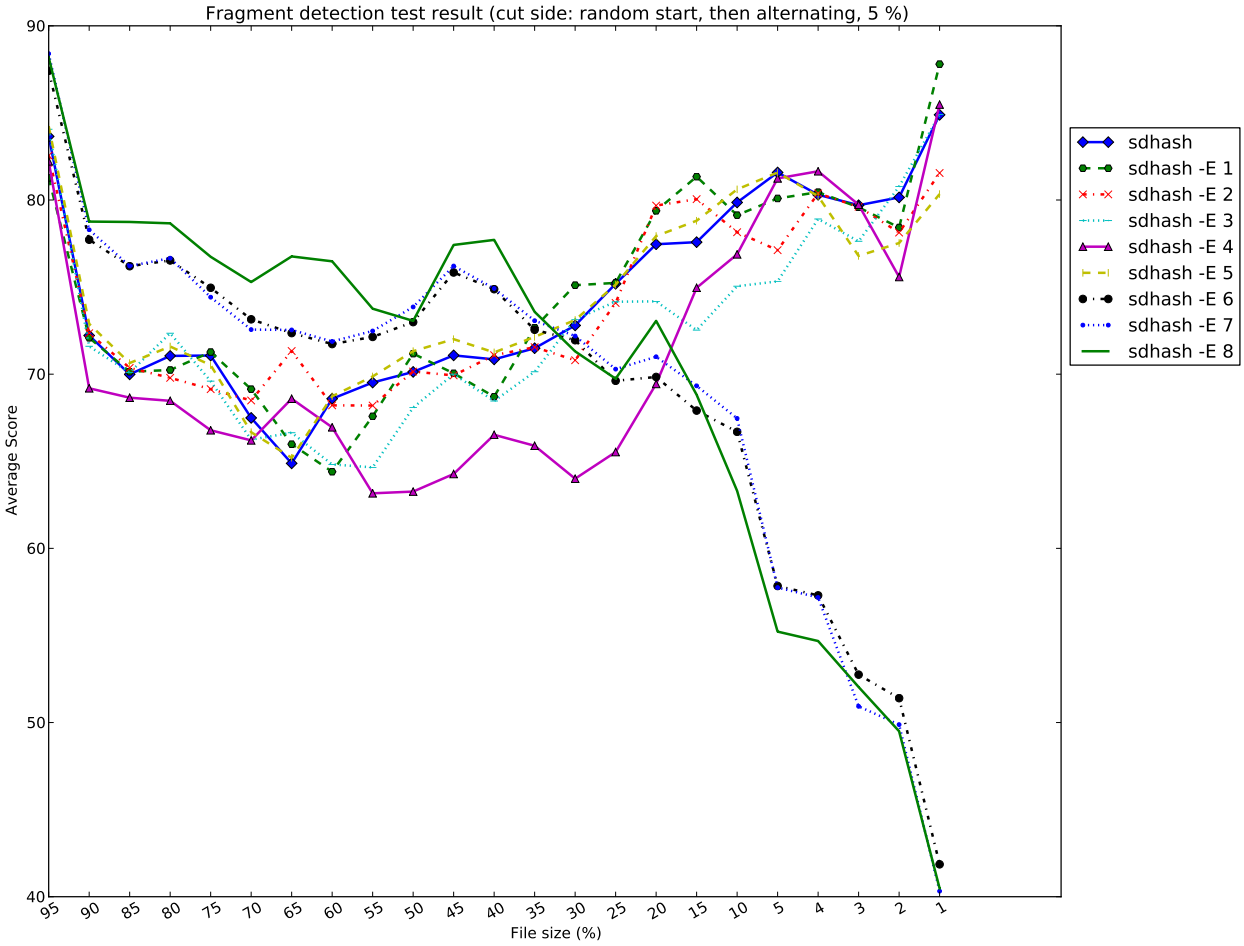


Figure 4.39: Average similarity scores for comparison of files against copies of themselves with slices removed from alternating ends, measured using 9 distinct *entropy rank* tables (slice size = 5% of file size until 5% remains, then 1%). `sdhash` uses Roussev's settings.

to decisively recommend one over another. One point of interest, however, is that this is the only test in the experiment for which ENT_0 and ENT_5 diverge. This occurs only briefly in Figure 4.38, between fragment sizes of five and one percent, but is more pronounced in Figure 4.39. This suggests that the non-random data does in fact contain a small number of low entropy features that ENT_5 is discarding. Because the fragment detection test will give a containment score for smaller and smaller fragments of non-random data, it is reasonable that small differences will become more pronounced.

4.4 Summary of Results

The highlights of our results are as follows:

1. Because our tests operate in continuous mode, the signature compression rate described in Section 4.1.1 provides a useful indicator of the feature selection rate of a given parameterization. This in turn allows one to predict many properties of its output. Low compression rates are associated especially with higher resistance to random noise but weaker fragment detection.
2. In general, *sdhash*'s expected or ideal behavior for each of the various tests is a function of the relative sizes of the files being compared.
    - Tests that compare similar-sized objects with varying degrees of common material are best interpreted as measuring commonality between objects. Ideal performance on this tests takes the shape of a smooth curve that goes to zero when common material is not present and shows maximum contrast between differing degrees of common material.
    - Tests that involve objects of different sizes and do not decrease the proportion of common material to dissimilar material in the smaller object are best interpreted as measuring containment. Ideal performance on containment tests takes the shape of a horizontal line indicating that the amount of material from the smaller object that is contained in the larger remains constant.
3. The properties of the data used in each test can have a significant impact on the outcome. In particular, randomly generated data has different properties from non-random data and this must be taken into account when designing tests and interpreting their results.
4. In regard to *sd score scale*, we can confidently recommend that a value of .1 be adopted, as this improves contrast in the commonality tests and consistency in the containment tests while still ensuring that dissimilar objects receive a zero score.
5. With respect to *popularity threshold* and *popularity window*, the best settings we found were 16 for both. However, because of their close relationship we recommend further experiments using a combinatoric approach.
6. Though we were only able to scratch the surface of possible variations in the *entropy rank* table, analysis of this parameter's space demonstrates that different choices may be better suited to different data, and moreover that testing these choices may provide a viable tool for examining the properties of a data set. Conversely, further research into the empirical properties of common file types may offer valuable insight into this parameter's behavior.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusions and Future Work

The goal of approximate matching algorithms is to create a mapping between the low level properties of binary data and the high level correlations that we recognize as similarity. This mapping is extraodinarily complex. Measuring the degree of similarity between arbitrary objects in such a way that corresponds with intuitive expectations requires the reproduction of an empirically determined network of high level relationships that is both poorly defined and subject to change. For this reason, we argue that it is unlikely any purely mathematical or logical determination of equivalence will be sufficient. Rather, a successful algorithm will incorporate its own empirical components, obtained in part from experimentation, iterative refinement, and analysis of targeted data.

This thesis endeavors to explore and refine a selection of the empirically derived parameters of *sdhash* while clarifying their relationship to the high level concept of similarity. To this end, we propose dividing the task of measuring similarity into two separate and more focused measurements indicating degree of containment and commonality. Our goal is to preserve the basic operation of the algorithm while making its output more intuitive and informative. Having presented the relevant background, experiment design and analysis of results in previous chapters, we turn now to avenues for future work and a summary of our contributions.

5.1 Future Work

In the course of our investigation of the *sdhash* parameter space, a number of opportunities for future work arose. Some of these were a consequence of the need to keep within the project scope; others were suggested by results of our analysis. The areas that we consider to have the highest priority include:

1. A comparison between continuous and block modes
2. A revisiting of our tests using a large sample size
3. Further investigation of the entropy rank table parameter
4. Examination of the empirical properties of our sample data, especially with respect to the distribution of H_{norm} values
5. Testing of additional parameters

In the sections that follow, we sketch a brief overview of each of these projects.

5.1.1 Block Mode

As described in Section 2.8.3, block mode's feature selection and signature creation methods differ significantly from those of continuous mode. By tying each Bloom filter to a specific block of data in the original object, this mode combines the tool's entropy-based triggers for feature selection with fixed cut points similar to sector hashing. In doing so, it introduces the possibility of unsaturated or even empty Bloom filters, which significantly alters the scores assigned to similar data. Additionally, block mode increase the maximum number of elements inserted in each filter to 192, and treats this number as a ceiling upon which it moves to the next block, even if the filter is filled early on. This allows potential features later in the block to be masked by features early on.

The implications of these changes are complex and could easily be influenced by variations in the test data. A side-by-side comparison of the two modes across the FRASH testing suite would contribute to the understanding of the effects of these alterations. This could be achieved simply by treating the different block size settings as a parameter. Preliminary results may also suggest further useful combinations for study, such as the relationship between popularity threshold or popularity window size and rate at which Bloom filters become saturated.

5.1.2 Sample Size

Our observations throughout all test results indicated that the properties of the test data often had a major impact on the results. We believe this to be an inherent challenge of work on approximate matching. Possible solutions include using a larger data set or multiple targeted data sets with some common property of interest. An alternative approach, discussed in Section 5.1.4, is to perform a more extensive analysis on the properties of the test data in order to understand what kinds of bias it may be introducing.

Increasing the size of the data set is the easier of the two approaches. A potential obstacle, however, is speed. Depending on the desired number of samples and the number of algorithms or parametrizations tested, several of the tests in the FRASH framework can take days to run using only the 85 files (each less than 1 MiB in size) in our data set. A first step in resolving this issue is to further modify our framework so that each test in FRASH is run as a separate process, with data stored in non-conflicting temporary directories in order to permit an arbitrary number of tests to run at once across a cluster.

5.1.3 Entropy Rank Tables

The results from our experiments with entropy rank tables indicate potential for follow-on investigations. In particular, the continued successful operation of *sdhash* when we replaced Roussev's empirically derived table with arbitrary values calls into question the role of these values in identifying features. ENT_4, which assigned values at random, is especially persuasive in this regard. Reproducing some of Roussev's own tests—such as those described in the real data study he performs as part of his comparison with *ssdeep*—with alternative tables may help to determine the actual impact of these choices, if any.

In addition, since the most dramatic results in the entropy rank experiments arose from changes in the boundaries of the null values, more tests varying the number and location of these value would provide useful information regarding which H_{norm} ranges are indispensable. As an example, eliminating the null values from the table altogether would provide insight into their effectiveness at reducing so called "weak" features. Alternatively, one could take a more fine grained approach than simply discarding features within certain H_{norm} ranges by combining the *entropy rank* table with a dynamic *popularity threshold*, that requires high entropy features to have a higher popularity before they are selected, while lowering it for low entropy areas. This would cause the algorithm to select a greater number of features from data regions that are associated with identifying data.

5.1.4 Empirical Investigation of Feature Distributions

The question raised in the previous section regarding appropriate H_{norm} to H_{prec} mapping draws draws attention to a more general problem. Throughout testing, results and analysis frequently highlighted the potential influence of the distribution of H_{norm} scores in the underlying data. Roussev's work has contributed to an understand of the typical frequency of these scores [15]; his study is based on 128-byte features, however.

Furthermore, no work has yet been done to describe typical arrangements of H_{norm} values in data. As we have argued in Section 3.2.5, this layout may strongly influence the feature selection process. An extensive study of typical empirical properties of common file types or drive contents may reveal trends that could inform the development of approximate matching algorithms.

5.1.5 Additional Parameters

Among the many possible parameters that could be explored, several stand out as especially interesting, but requiring significantly more extensive reworking of *sdhash*'s code. Perhaps the most obvious of these is feature size. Roussev has considered 128-byte features; knowing the impact of modifying the feature size is an important driving factor for choosing the algorithm's design. Unfortunately, because much of the code is optimized specifically for 64-byte features, varying this parameter across a range is not as simple as adding an option to change a scalar value (for example, provided that the entropy rank table is adding value to the selection process, a new table would have to be developed for each potential size).

Though unsuitable for practical uses, another interesting design question could be investigated by altering the algorithm to avoid the use of Bloom filters altogether, instead performing a direct comparison of the features themselves. A considerable advantage of this undertaking would be the enhanced ability to gather inspect the selected features themselves.

Finally, additional possibilities for exploration include testing large-to-small comparisons and performing combinatoric tests of feature threshold and popularity window size.

5.2 Contributions

Our contributions consist mainly in recommendations for the improvement of the FRASH testing framework and *sdhash*. Several of these have been adopted already. We provide a full description of each.

5.2.1 Modification to the *sdhash* Popularity Score Routine

As noted in Section 4.1.1, a large number of features with H_{norm} scores that map to null H_{prec} scores appears to slow the signature generation process. We believe this to be the result of a minor bug in the popularity scoring routine, resulting from an ambiguity caused by the use of 0 as the null-value indicator. This bug is unlikely to have an impact on typical data unless other aspects of the algorithm are changed (such as the mapping of H_{norm} scores to nulls). We believe it can be resolved with trivial changes, however, and doing so may have the added benefit of adding a slight speed improvement to signature generation in general. Revised code resolving the problem has been contributed back to the developer.

5.2.2 Parameter Choices

Based on our analysis, we recommend the adoption of a value of 0.1 for the sd score scale parameter. The current setting is 0.3; however, we encountered no problems using the lower value and observed improved performance across the board: the commonality tests showed increased contrast and the containment tests maintained better consistency in the face of alignment shifts and fragmentation.

Although our study of the effects of changes in the popularity window size and popularity threshold suggest that there may be some advantage to setting both equal to 16 (this appears to be the best among the values tested) we refrain from making this recommendation until further work can be done checking each combination of the two values.

With respect to the entropy rank table, we note that the algorithm's performance does not appear to depend crucially on the values provided by Roussev. If additional testing confirms that multiple settings yield equal performance, it may be advantageous to deliberately vary the table across different organizations to prevent any systematic attempt to defeat *sdhash* by intentionally adding chunks of data that are always selected as features. A disadvantage of this approach would be that SDBFs could not be shared between organizations with different tables.

Finally, we emphasize that our exploration of parameters in the current implementation of *sdhash*, which is tuned towards containment, would also be relevant to a version that allowed the user to request a commonality score under both of the implementations that we suggest for such a score in Section 5.2.4.

5.2.3 FRASH Testing Framework Recommendations

A possible critique of the FRASH testing framework is that it uses artificial experiments instead of tagged data. We argue that this is a strength. Tagging data for similarity would first require agreement as to how humans should score the similarity of two objects. Even when the tagged is reduced to a binary classification, such a consensus is tenuous and may have the drawback of appearing to demonstrate ground truth when no such guarantee is available. Providing a comprehensive description of the natural language meaning of the term is a daunting project, and lobbying for widespread adoption of such a description is problematic. Instead, we advocate a bottom-up approach that begins first the the development of a useful metric and then demonstrates what it can tell us about a given object or set of data.

That said, further modifications to FRASH could enhance its ability to offer a generic testing

framework for comparing multiple approximate matching algorithms to each other, or different parameterizations of the same algorithm against itself. We offer some comments based on our use of the tool, beginning with general suggestions, and proceeding to recommendations for some of the specific tests before ending with notes regarding some subtle points of testing approximate matching functions.

General Suggestions

To permit effective evaluation of multiple approximate matching algorithms, it is important to note that scores from different algorithms are unlikely to be directly comparable to each other. Until some standard metric is developed, we are forced to add an addtional layer of interpretation before performing a comparison. One method of accomplishing this would be to use score cutoffs to convert the output into a binary classification decision which we can then use to create a confusion matrix. Another method is to use the algorithms to create relative orderings of object pairs, and examine the differences in the rankings.

In our experience using FRASH to test parameters, we have found that a more useful approach is to disregard the question of algorithm correctness in favor of more descriptive data showing the patterns in its output across the samples produced by each test. In this regard, a significant limitation of FRASH is that it is designed to give its output in terms of ASCII tables on the terminal. This formatting choice imposes restrictions on the number of samples taken for each test, which is frequently set too low by default to give a picture of algorithm behavior with confidence. Outputting to comma separated values, XML, or other data that can be easily parsed and graphed would be preferable.

Also relating to the issue of default values are the step sizes set for the various tests. We demonstrated in Section 4.3.1 that default settings for the alignment test prevent the observation of crucial trends in *sdhash* behavior. We acknowledge that this may not be the case across all matching algorithms. Some option to set appropriate step sizes and maximums without the need to modify the source code would greatly enhance the framework's utility, however.

Finally, as we have argued in Section 3.1.3, it is important that all tests in the framework have output-independent end conditions. The reasoning behind this assertion includes greater robustness in the face of poorly designed or mis-tuned algorithms; it has been presented to the developers and already adopted in FRASH 2.0. Here we emphasize one additional point that became apparently after examining our results: because the behavior of the score is frequently non-linear and in fact markedly periodic, it is crucial to be able to continue to test after a zero

value to ensure that the additional activity of interest (in this case non-zero scores) is not inadvertently excluded.

Comments on Fragment Detection and Random-noise Resistance Tests

As noted in our initial description (see Section 3.1.3), the fragmentation test compares an object with a standalone section of itself. This is an unusual detection scenario, since fragments are generally found embedded in other data, and it has the side-effect of causing problems from *sdhash* once the size of the file drops below 512 bytes. It would be useful to create an alternative fragmentation test that is similar to the single-common-block test but embeds a fragment of similar material instead of a randomly generated block. Following Roussev in his comparison of *sdhash* and *ssdeep* [3], a test that embeds multiple fragments that have been reordered would also provide interesting data.

Although the random-noise test has been altered to report results in terms of average scores for better comparison with the other tests in the framework, its current design presents difficulties in estimating how much similar material remains (see Section 3.1.3 for relevant calculations). We believe the test could be modified to use only substitutions without significantly altering the results, though experimentation would be required to confirm this. If these substitutions were made by selecting random bytes without replacement, we would always be able to calculate how much of the file has been changed. This metric would be much easier to understand than the number of edits proportional to the total number of bytes in the file.

Subtleties of Testing Approximate Matching Algorithms

While analyzing test results and attempting to explain trends, we encountered a number of subtle issues that are important to note. This first of these appeared at the outset: test results can be dramatically influenced by the relative sizes of the inputs. This may be especially true of *sdhash* since it reports small-to-large scores. However, the challenge of handling size discrepancies is at the heart of current work in approximate matching, and a driving factor in algorithm development. Ignoring the effect of size differences can result in misleading generalizations.

Second, not all data can be treated equally. In particular the properties of randomly generated data are distinctive, and can often produce very different behavior than a similar test using non-random data. In some cases, of course, randomly generated data is useful (particularly as a baseline for dissimilar material); in other cases it may not be avoidable. Regardless of the circumstance, it is important to acknowledge the properties of the test data used and factor these in when determining potential biases of the tests.

5.2.4 Commonality Score

Our final contribution is the proposal to adapt *sdhash* to allow the user to request a commonality score instead of a containment score. There are a number of possibilities for implementing this change without need for extensive modification to the code. The simplest approach is to reverse the default behavior and report the similarity score produced by comparing the larger object to the smaller. Doing so would cause the tool to give a result that could be consistently used as an indicator of the ratio of similar to dissimilar material.

Like the current small-to-large scores, a large-to-small score has a blind spot. Dissimilar material could be added to the smaller of the two files without producing any change in the score. Whether this is problematic depends on how the concept of "similarity" is understood by users of the tool. To give an analogy, consider two comparisons: a book compared to one of its own chapters, and two books with a single chapter in common. Assuming all chapters and both books are the same size, should one of these pairs receive a higher score? If not, then large-to-small comparison is a sufficient measure of similarity. If the two books with a common chapter are considered less similar than one of the books compared to the chapter alone, some further alterations will be needed. The most practical approach to this decision is to allow the answer to be driven by common use cases and efficiency considerations.

A potential advantage of using a scoring system in which adding dissimilar material to the smaller of the two objects reduces the score is that this behavior might be more intuitive for analysts, as it meets the expectation that increasing similar material always increases the score whereas increasing dissimilar material always decreases it. Though the implementation is slightly more complex, the simplicity of such a rule reduces the likelihood that the algorithm's behavior will be misunderstood, thus making its output more accessible. Furthermore, a scoring system with these properties could be derived from the system *sdhash* currently uses without need for major modification. A possible solution would be to use an average of the small-to-large and large-to-small scores.

Finally, we note that use case for commonality is not yet as clearly stated as the case for containment. One possible application is the task of comparing a collection of irregular objects or groups of data, like collected media. A containment score cannot distinguish between two very similar objects and an object compared with a fragment.

Another potential application is the ability to correlate data in a holding without a sample of

target data. We believe more uses will be developed when the capability is available, perhaps including combinations of the two types of scores: for example, a commonality score could be used to find two objects with a large amount of common material, followed by the use of a containment score to search for all the objects that contain fragments of that same material.

Even if the utility of one metric turns out to outweigh that of the other, the subdivision of the general term similarity has the advantage, at the least, of creating a distinction between commonality and containment. This in turn helps to set the correct expectations for *sdhash*'s output, thus advancing our overarching goal of bringing its interpretation in line with intuition.

In pursuit of this objective, we present a detailed exploration of the *sdhash* parameter space, and a thorough characterization of the algorithm's behavior. We leverage this characterization to define a distinction between notions of containment and commonality, which permits a clarification of criteria by which approximate matching algorithms can be judged. Our proposals for improving the FRASH testing framework are based on these criteria, as is our recommendation of a new *sd score scale* value.

Each of these contributions represent progress toward the larger project of reconciling a numerical scoring system—based solely on the binary representation of data—with expectations driven by the natural-language notion of similarity. This undertaking has a direct application to forensic analysis, and promises improved capabilities for data triage and attribution.

THIS PAGE INTENTIONALLY LEFT BLANK

REFERENCES

[1] V. Roussev, "Data fingerprinting with similarity digests," in *Advances in Digital Forensics VI*, ser. IFIP Advances in Information and Communication Technology, K.-P. Chow and S. Shenoi, Eds. Springer Boston, 2010, vol. 337, pp. 207–226. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-15506-2_15

[2] F. Breitinger, G. Stivaktakis, and H. Baier, "Frash: A framework to test algorithms of similarity hashing," in *Proceedings of the 13th Digital Forensics Research Conference*. Monterey, CA: DFRWS'13, 2013, to appear.

[3] V. Roussev, "An evaluation of forensic similarity hashes," *Digital Investigation*, vol. 8, Supplement, no. 0, pp. S34 – S41, 2011, the Proceedings of the Eleventh Annual DFRWS Conference 11th Annual Digital Forensics Research Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287611000296

[4] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[5] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[6] R. C. Merkle, "Secrecy, authentication, and public key systems." Ph.D. dissertation, Stanford University, Stanford, CA, 1979.

[7] I. B. Damgård, "A design principle for hash functions," in *Advances in Cryptology–CRYPTO'89 Proceedings*. New York: Springer, 1990, pp. 416–427.

[8] R. C. Merkle, "One way hash functions and des," in *CRYPTO*, 1989, pp. 428–446.

[9] D. White, "Nist national software reference library," 2005. [Online]. Available: http://www.nsrl.nist.gov/

[10] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, Supplement, no. 0, pp. 91 – 97, 2006, the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287606000764

[11] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks, "Distinct sector hashes for target file detection," *IEEE Computer Society*, vol. 45, no. 12, pp. 28–35, 2012.

[12] M. Rabin, "Fingerprint by random polynomials," Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, Tech. Rep., 1981.

[13] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, Australian National University Canberra, Canberra, 1999.

[14] A. TRIDGELL, "Spamsum. readme (2002)." [Online]. Available: http://samba.org/ftp/unpacked/junkcode/spamsum/README

[15] V. Roussev, "Building a better similarity trap with statistically improbable features," in *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*. Honolulu: IEEE, 2009, pp. 1–10.

[16] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *Digital Investigation*, vol. 6, pp. S2–S11, Sep 2009. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2009.06.016

[17] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic clustering of the web," *Computer Networks and ISDN Systems*, vol. 29, no. 8, pp. 1157 – 1166, 1997, papers from the Sixth International World Wide Web Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169755297000317

[18] V. Roussev, "Scalable data correlation," in *Eighth Annual IFIP WG 11.9 International Conference on Digital Forensics*, Pretoria, South Africa, Jan 2012.

[19] V. Roussev and C. Quates, "Content triage with similarity digests: The m57 case study," *Digital Investigation*, vol. 9, Supplement, no. 0, pp. S60 – S68, 2012, the Proceedings of the Twelfth Annual DFRWS Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287612000370

[20] V. Roussev, Y. Chen, T. Bourg, and G. G. R. III, "md5bloom: Forensic filesystem hashing revisited," *Digital Investigation*, vol. 3, Supplement, no. 0, pp. 82 – 90, 2006, the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287606000740

[21] S. Garfinkel, "Digital forensics research: The next 10 years," *Digital Investigation*, vol. 7, Supplement, no. 0, pp. S64 – S73, 2010, the Proceedings of the Tenth Annual DFRWS Conference. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1742287610000368

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia
2. Dudly Knox Library
   Naval Postgraduate School
   Monterey, California